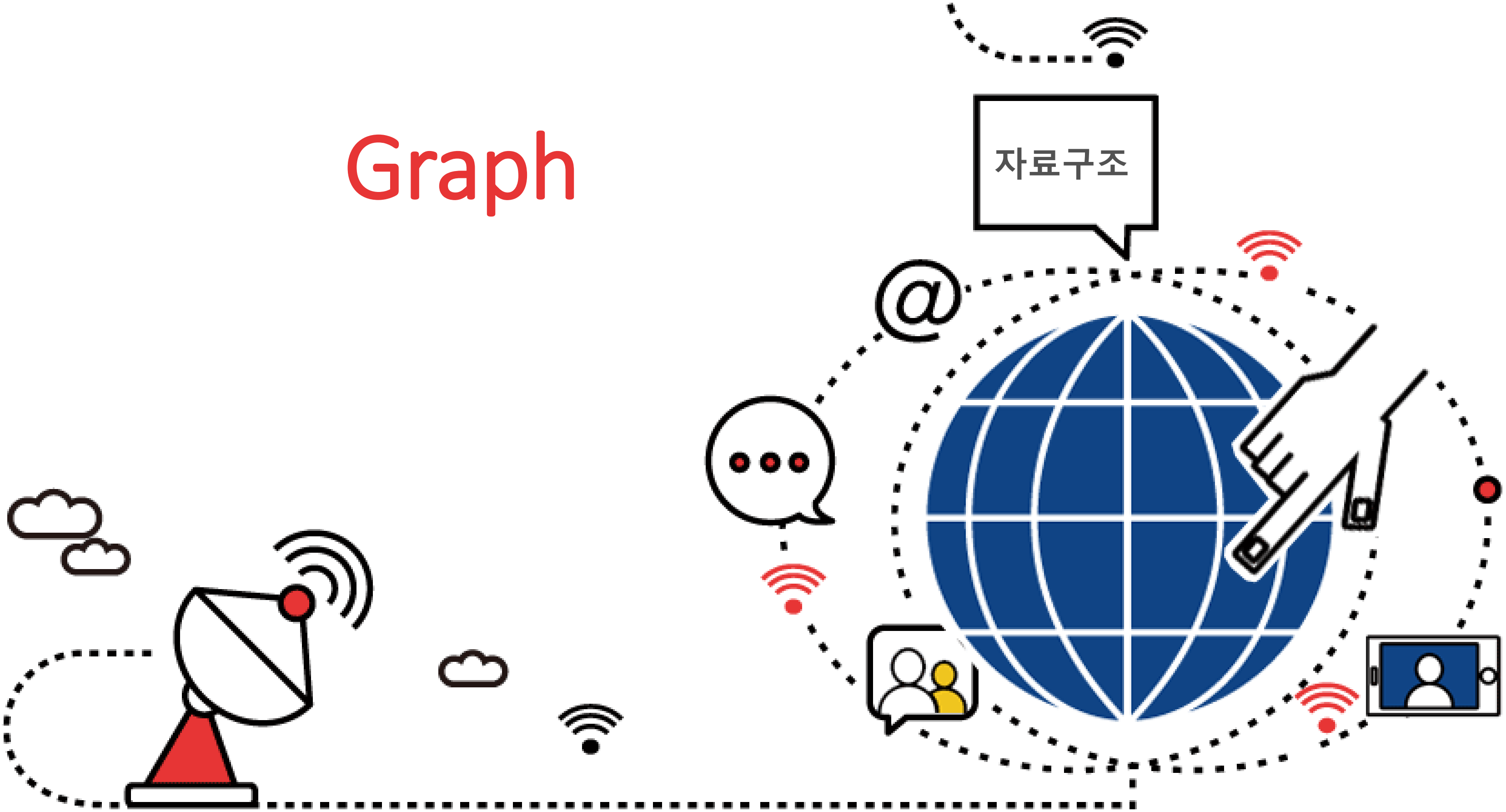
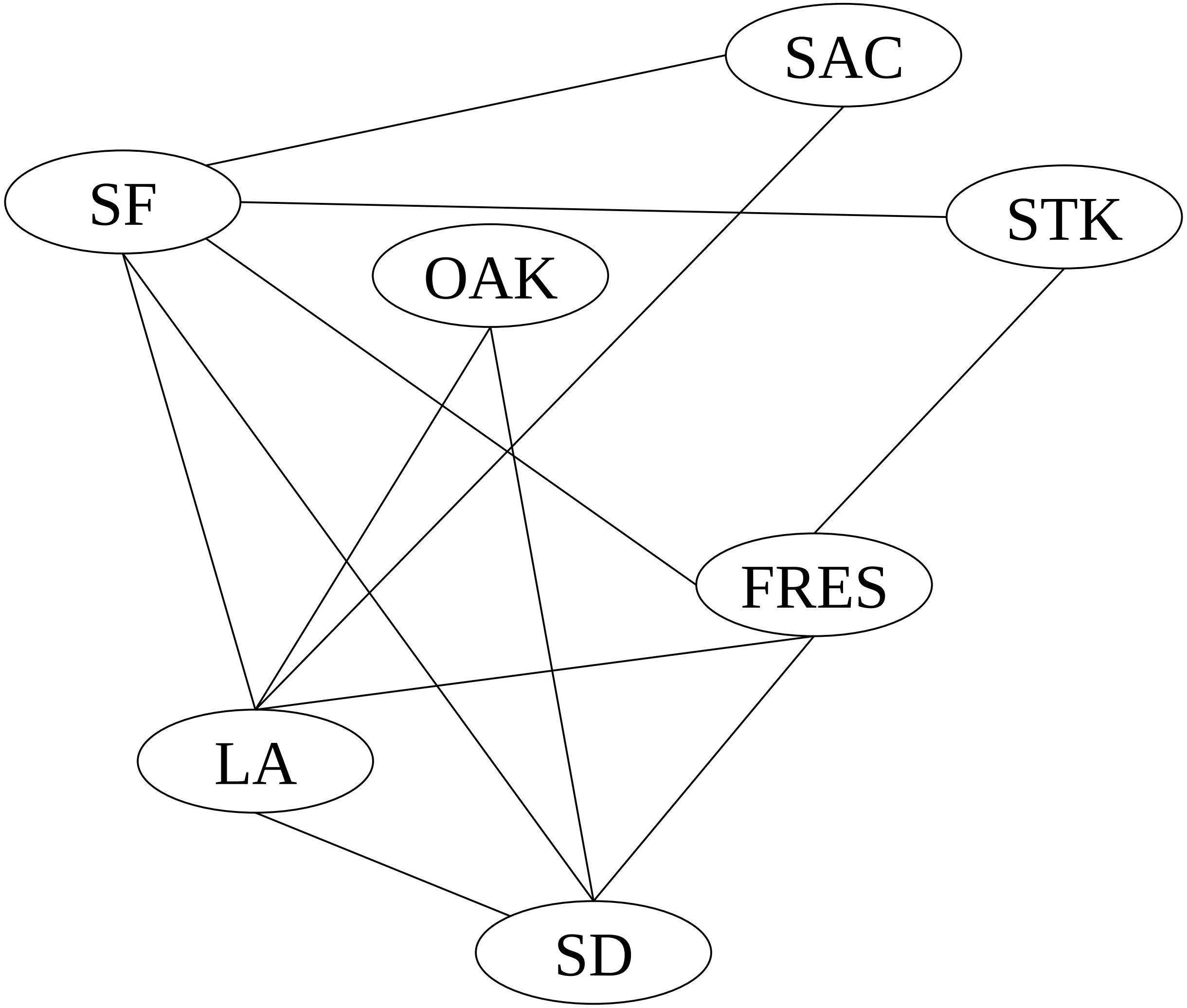
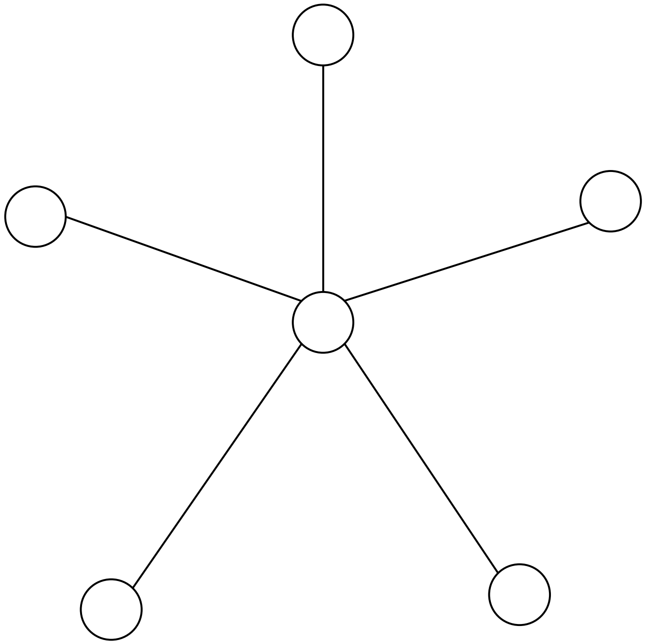
Graph



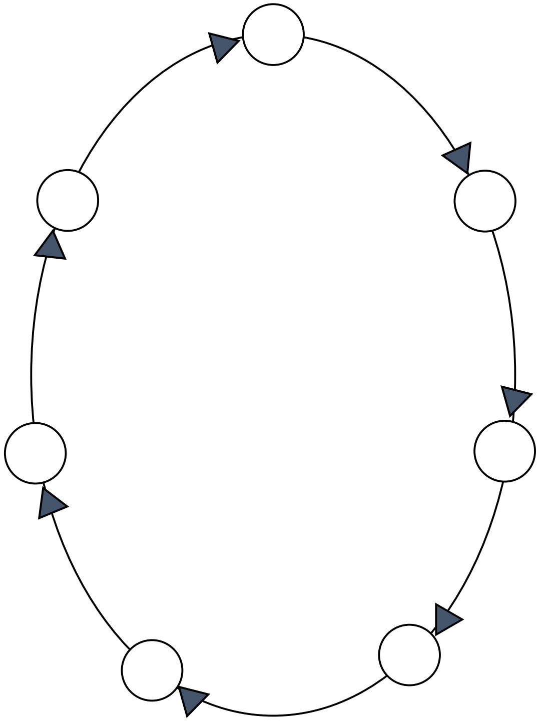
그래프 예시 - 공항



그래프 예시 - 네트워크



(a) A star network

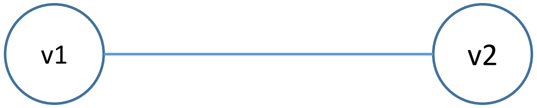


(b) A ring network

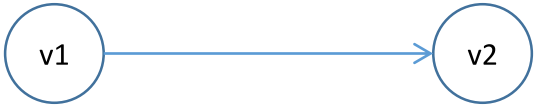
Graph

Graph 란?

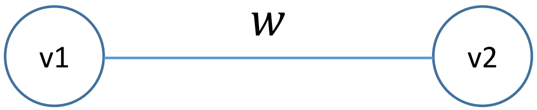
- 정점(Vertex)과 간선(Edge)으로 이루어진 자료구조
- 간선의 종류에 따라 그래프 종류가 달라진다
  - Undirected Graph : 간선이 방향 X



  - Directed Graph : 간선이 방향을 가리킴

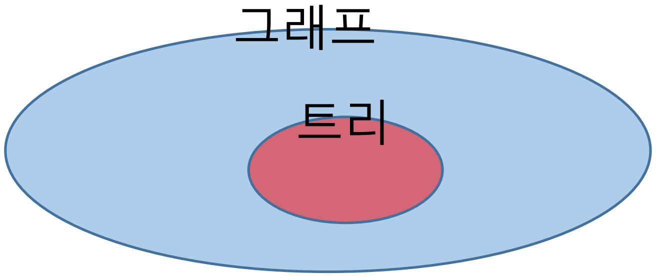


  - Weighted Graph : 간선에 가중치가 있음



Graph vs Tree

그래프	트리
노드와 노드를 연결하는 간선들의 집합	Cycle이 없는 Connected Graph
Cycle 존재 가능	Cycle 존재 불가능
두 정점 사이에 여러 개의 경로 존재 가능	두 정점 사이에 반드시 1개의 경로 존재
간선의 수는 그래프에 따라 다름	간선의 수 = 정점의 수 -1



그래프 ⊃ 트리

Graph

- **Graph** $G = (V, E), n = |V|, m = |E|$
- **Weighted Graph** $G = (V, E, W)$
- **Graph의 표현방법(Representation)은 2가지**
  - Adjacency Matrix Representation(인접 행렬)
  - Adjacency List Representation (인접 리스트)

▪ no parallel edges ▪ no self-loops	Adjacency List	Adjacency Matrix
Space	$n + m$	n^2
v.incidentEdges()	deg(v)	n
v.isAdjacentTo (w)	min(deg(v), deg(w))	1
insertVertex(o)	1	n^2
insertEdge(v, w, o)	1	1
eraseVertex(v)	deg(v)	n^2
eraseEdge(e)	1	1

표현법의 차이점

인접 행렬

- 노드 i와 j의 인접(adjacent) 여부를 $O(1)$ 시간에 확인 가능
- 밀집한(dense) 그래프에서 효율적
- 구현이 쉽다

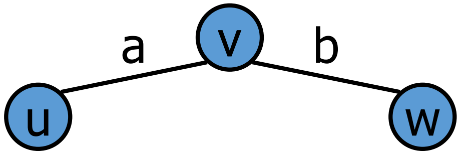
인접 리스트

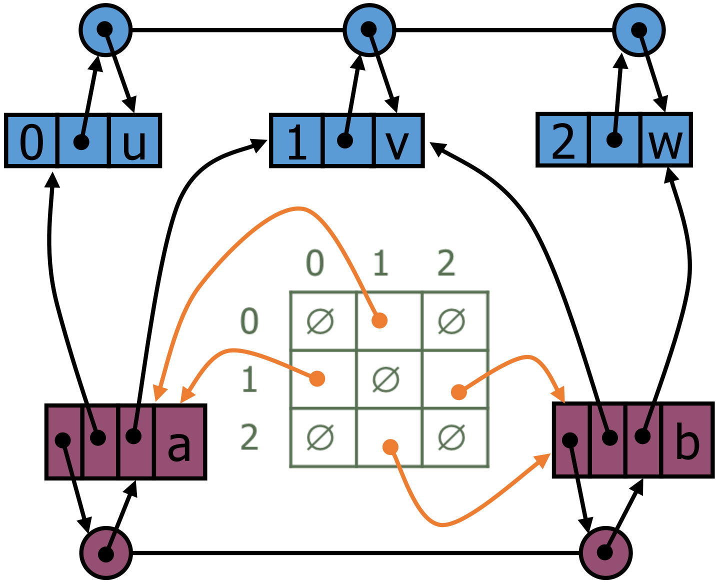
- 노드 v에 인접한(incident) 간선들의 정보를 $O(degree(v))$ 시간에 확인 가능
- 드문(sparse) 그래프에서 효율적
- 구현이 상대적으로 복잡하다

인접행렬 표현법

인접 행렬 표현법 (Adjacency Matrix Representation)

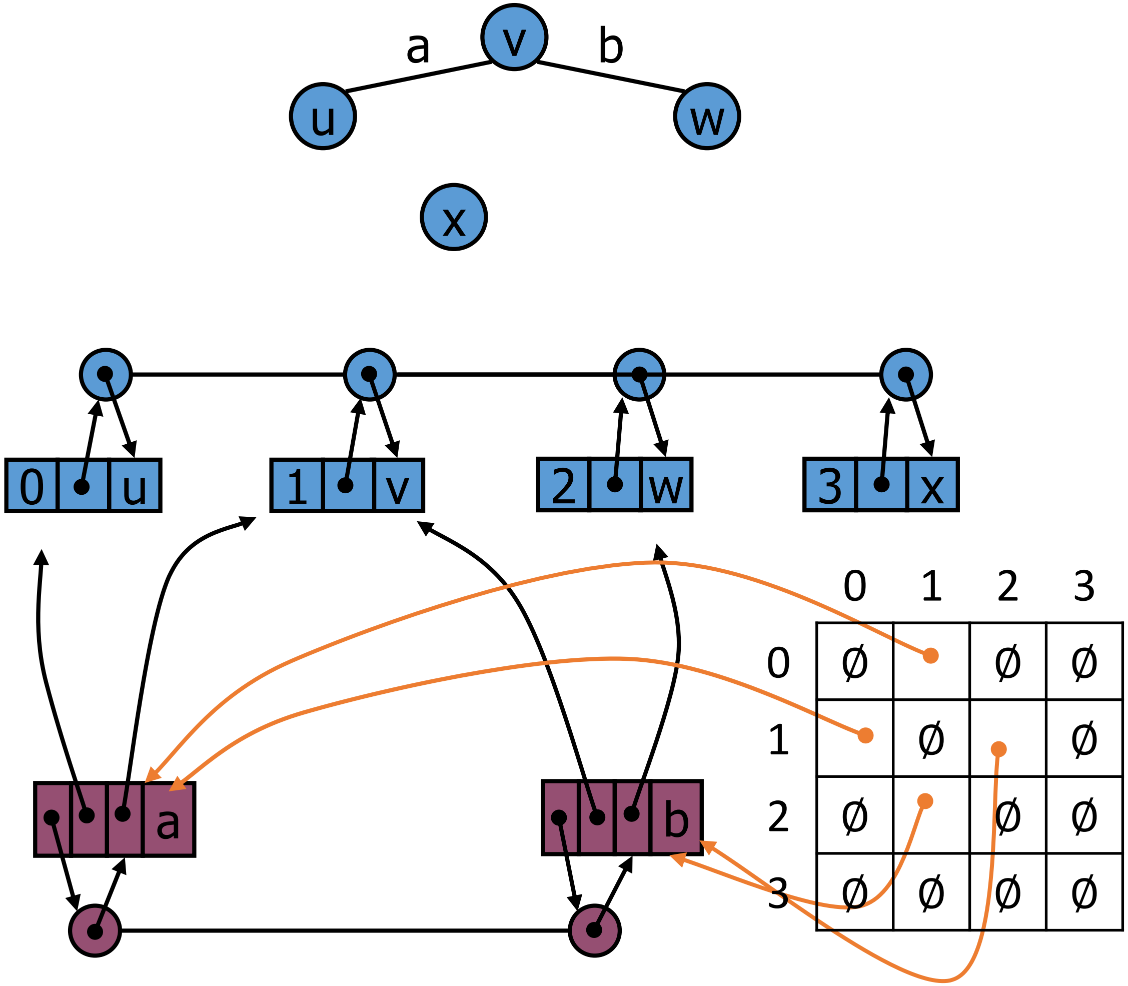
- Graph에 대한 인접 행렬은 $|V| \times |V|$의 행렬을 사용한다
- 노드 i와 j사이에 간선이 존재하면 간선의 pointer, 아니면 NULL
- Weighted Graph일 경우 간선에 weight 값을 저장한다





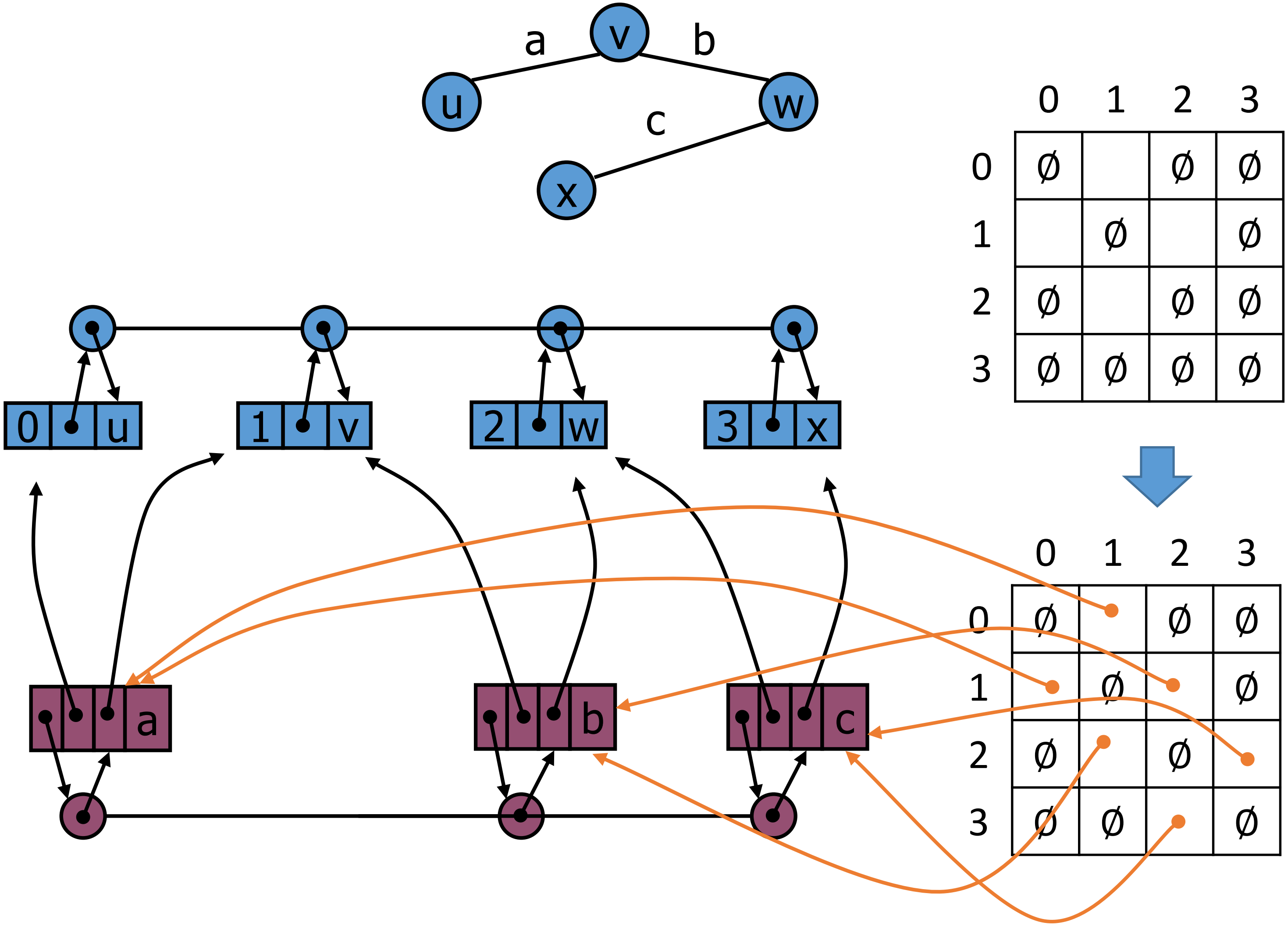
	0	1	2
0	∅	●	∅
1	●	∅	●
2	∅	●	∅

정점 삽입



	0	1	2	3
0	Ø	●	Ø	Ø
1	●	Ø	●	Ø
2	Ø	●	Ø	Ø
3	Ø	Ø	Ø	Ø

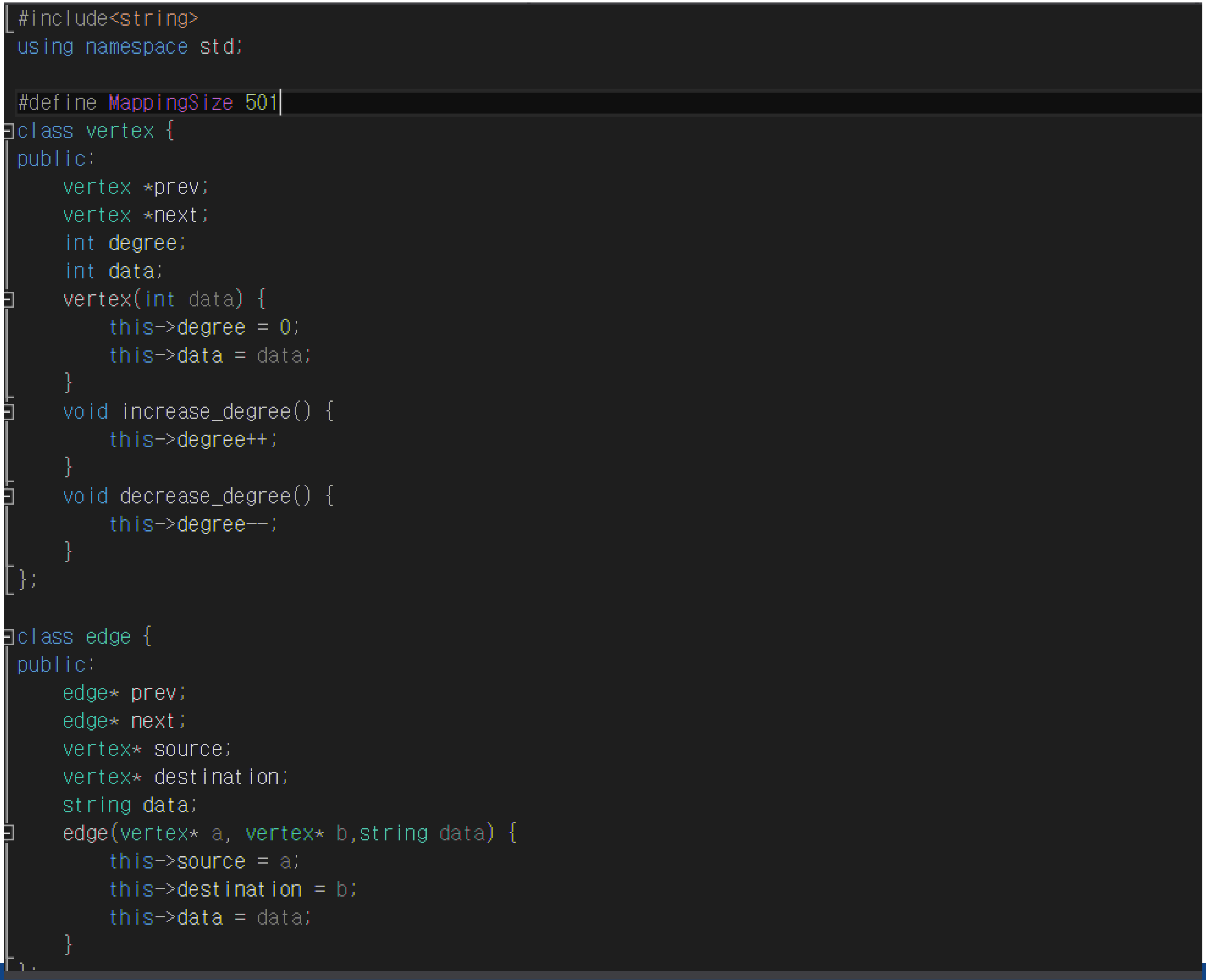
간선 삽입



	0	1	2	3
0	Ø		Ø	Ø
1		Ø		Ø
2	Ø		Ø	Ø
3	Ø	Ø	Ø	Ø

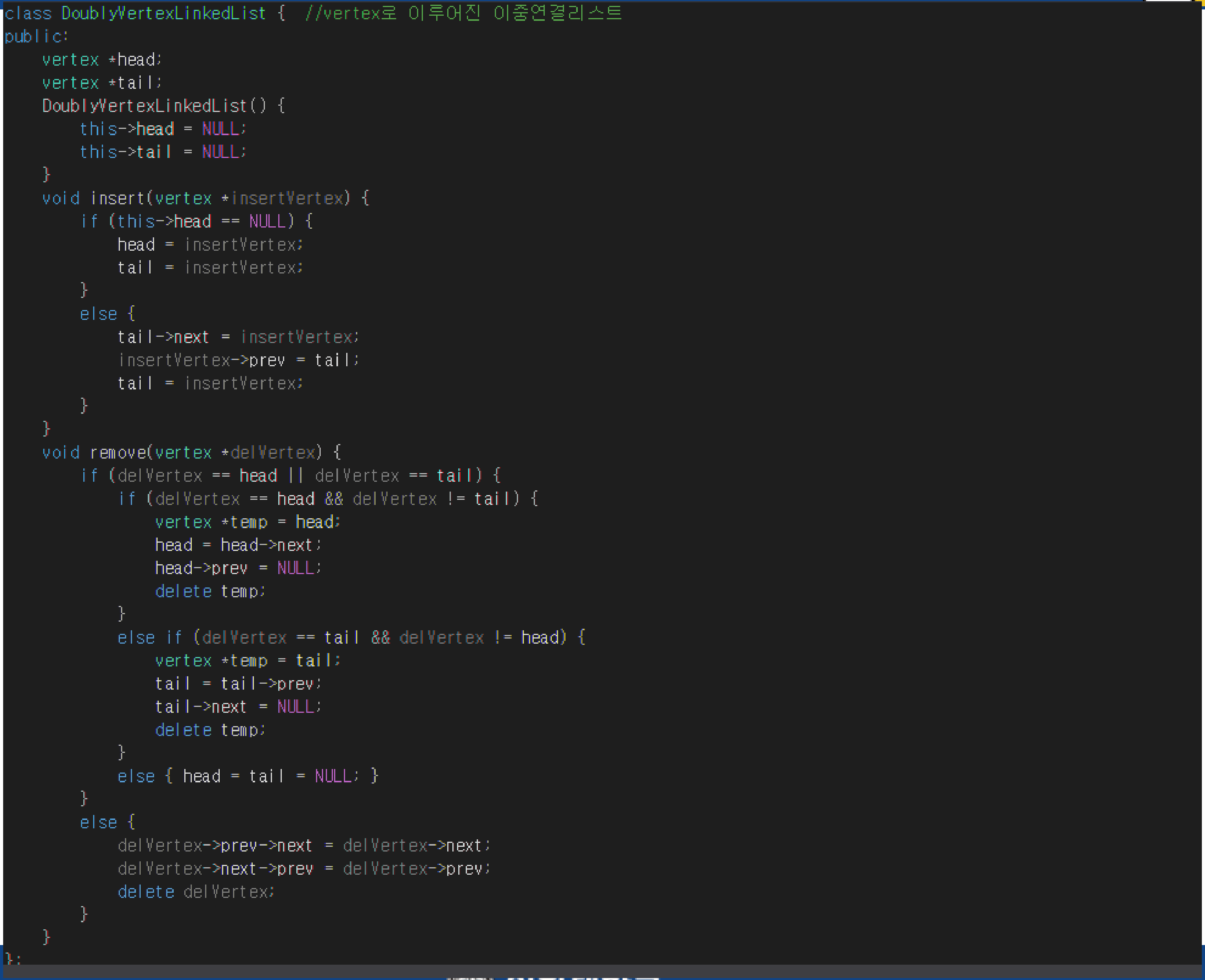
	0	1	2	3
0	Ø	●	Ø	Ø
1	●	Ø	●	Ø
2	Ø	●	Ø	●
3	Ø	Ø	●	Ø

코드(vertex, edge)



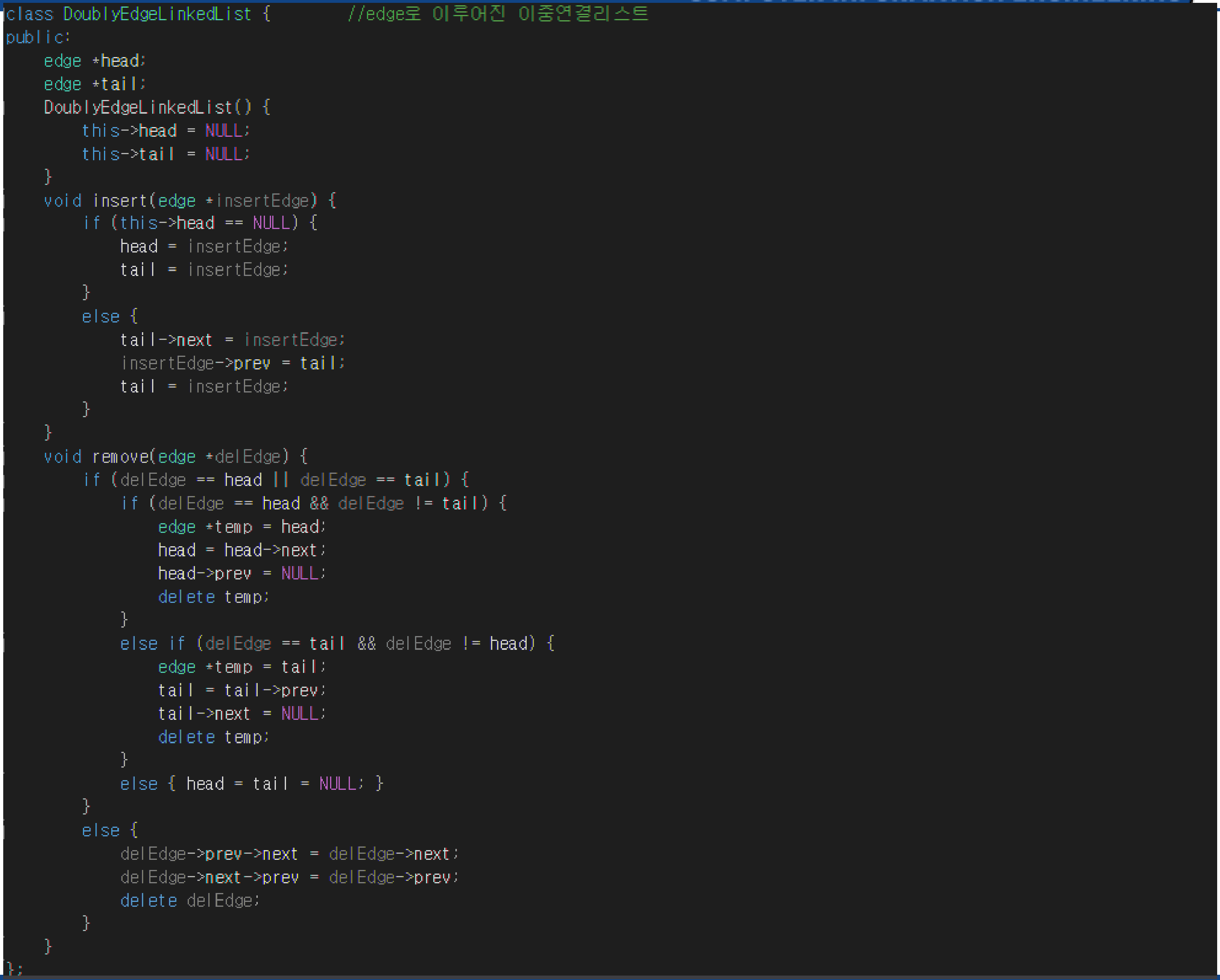
```
#include<string>
using namespace std;

#define MappingSize 501
class vertex {
public:
    vertex *prev;
    vertex *next;
    int degree;
    int data;
    vertex(int data) {
        this->degree = 0;
        this->data = data;
    }
    void increase_degree() {
        this->degree++;
    }
    void decrease_degree() {
        this->degree--;
    }
};

class edge {
public:
    edge* prev;
    edge* next;
    vertex* source;
    vertex* destination;
    string data;
    edge(vertex* a, vertex* b,string data) {
        this->source = a;
        this->destination = b;
        this->data = data;
    }
};
```

코드 (VertexList)



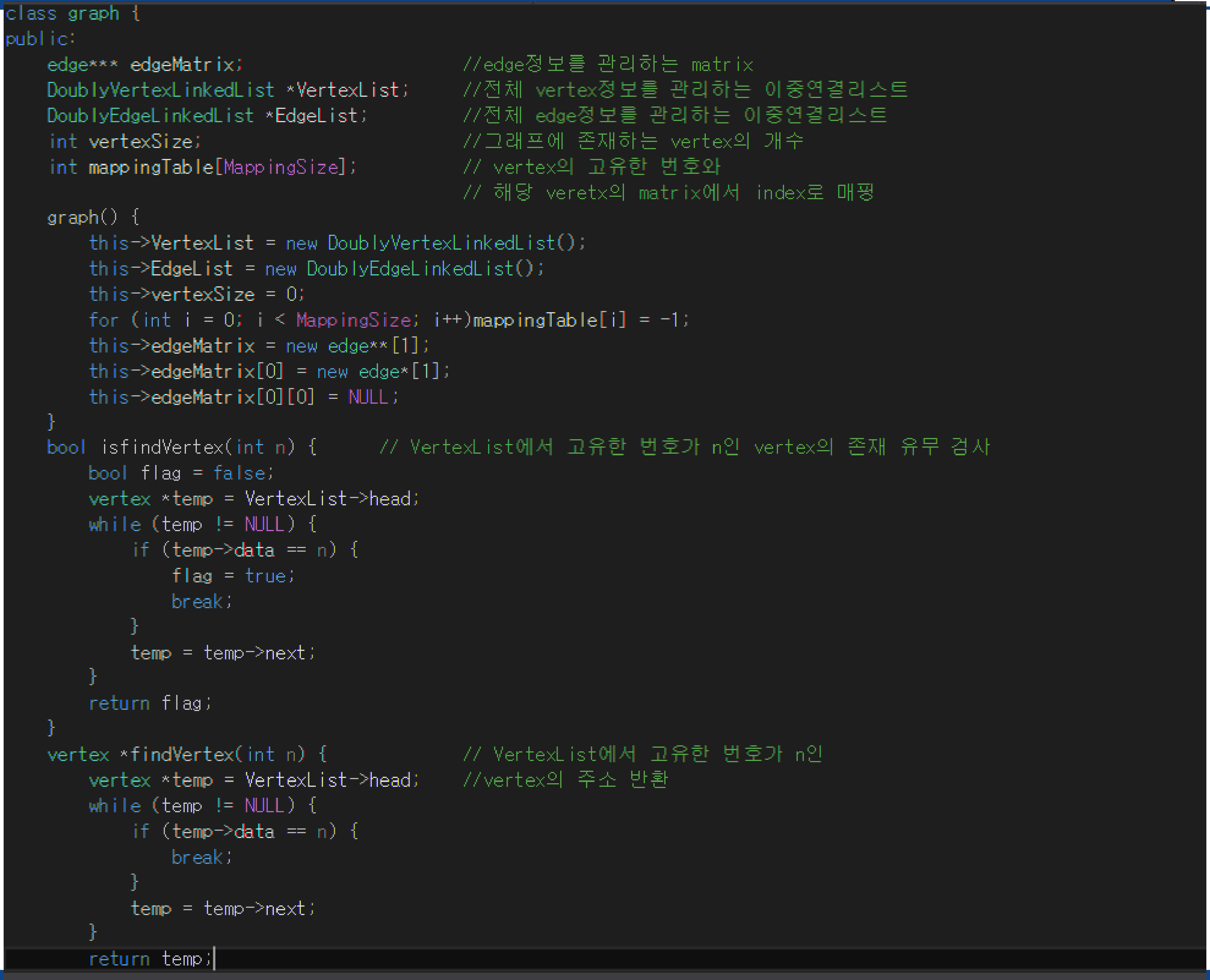
```
class DoublyVertexLinkedList {  //vertex로 이루어진 이중연결리스트
public:
    vertex *head;
    vertex *tail;
    DoublyVertexLinkedList() {
        this->head = NULL;
        this->tail = NULL;
    }
    void insert(vertex *insertVertex) {
        if (this->head == NULL) {
            head = insertVertex;
            tail = insertVertex;
        }
        else {
            tail->next = insertVertex;
            insertVertex->prev = tail;
            tail = insertVertex;
        }
    }
    void remove(vertex *delVertex) {
        if (delVertex == head || delVertex == tail) {
            if (delVertex == head && delVertex != tail) {
                vertex *temp = head;
                head = head->next;
                head->prev = NULL;
                delete temp;
            }
            else if (delVertex == tail && delVertex != head) {
                vertex *temp = tail;
                tail = tail->prev;
                tail->next = NULL;
                delete temp;
            }
            else { head = tail = NULL; }
        }
        else {
            delVertex->prev->next = delVertex->next;
            delVertex->next->prev = delVertex->prev;
            delete delVertex;
        }
    }
};
```

코드(EdgeList)



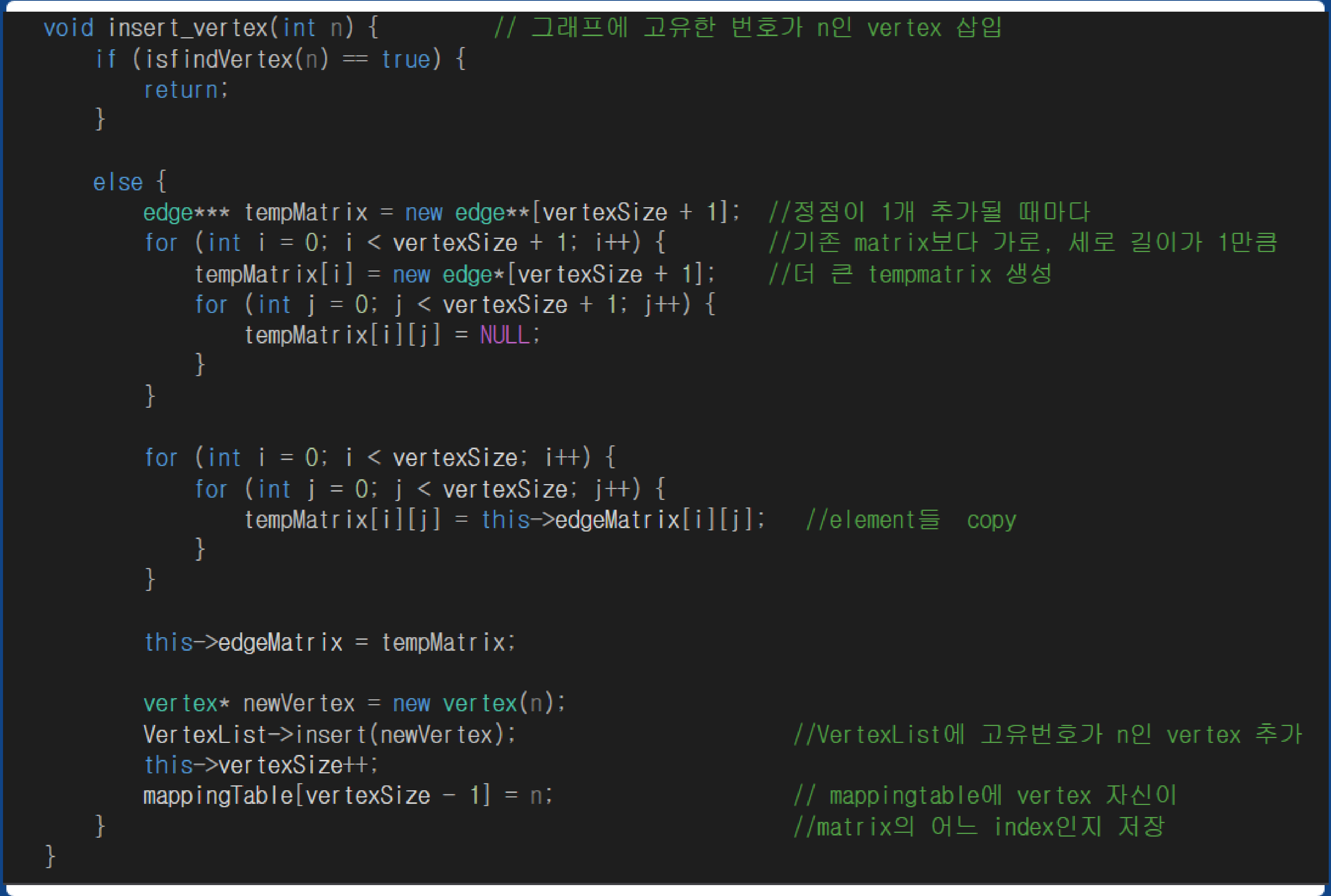
```
class DoublyEdgeLinkedList {        //edge로 이루어진 이중연결리스트
public:
    edge *head;
    edge *tail;
    DoublyEdgeLinkedList() {
        this->head = NULL;
        this->tail = NULL;
    }
    void insert(edge *insertEdge) {
        if (this->head == NULL) {
            head = insertEdge;
            tail = insertEdge;
        }
        else {
            tail->next = insertEdge;
            insertEdge->prev = tail;
            tail = insertEdge;
        }
    }
    void remove(edge *delEdge) {
        if (delEdge == head || delEdge == tail) {
            if (delEdge == head && delEdge != tail) {
                edge *temp = head;
                head = head->next;
                head->prev = NULL;
                delete temp;
            }
            else if (delEdge == tail && delEdge != head) {
                edge *temp = tail;
                tail = tail->prev;
                tail->next = NULL;
                delete temp;
            }
            else { head = tail = NULL; }
        }
        else {
            delEdge->prev->next = delEdge->next;
            delEdge->next->prev = delEdge->prev;
            delete delEdge;
        }
    }
};
```

코드 (Graph)



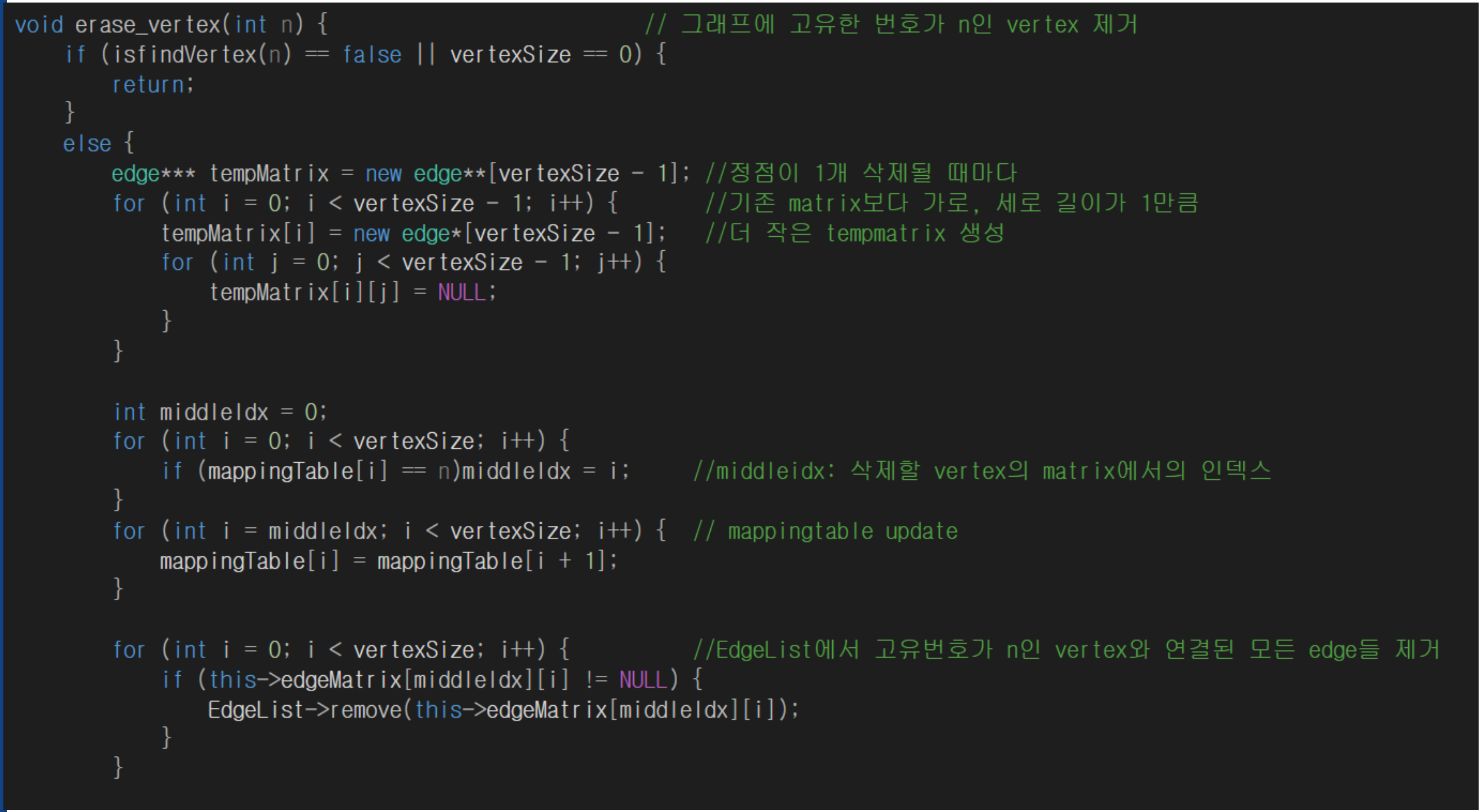
```
class graph {
public:
    edge*** edgeMatrix;                     //edge정보를 관리하는 matrix
    DoublyVertexLinkedList *VertexList;     //전체 vertex정보를 관리하는 이중연결리스트
    DoublyEdgeLinkedList *EdgeList;         //전체 edge정보를 관리하는 이중연결리스트
    int vertexSize;                         //그래프에 존재하는 vertex의 개수
    int mappingTable[MappingSize];          // vertex의 고유한 번호와
                                            // 해당 veretx의 matrix에서 index로 매핑
    graph() {
        this->VertexList = new DoublyVertexLinkedList();
        this->EdgeList = new DoublyEdgeLinkedList();
        this->vertexSize = 0;
        for (int i = 0; i < MappingSize; i++)mappingTable[i] = -1;
        this->edgeMatrix = new edge**[1];
        this->edgeMatrix[0] = new edge*[1];
        this->edgeMatrix[0][0] = NULL;
    }
    bool isfindVertex(int n) {      // VertexList에서 고유한 번호가 n인 vertex의 존재 유무 검사
        bool flag = false;
        vertex *temp = VertexList->head;
        while (temp != NULL) {
            if (temp->data == n) {
                flag = true;
                break;
            }
            temp = temp->next;
        }
        return flag;
    }
    vertex *findVertex(int n) {             // VertexList에서 고유한 번호가 n인
        vertex *temp = VertexList->head;    //vertex의 주소 반환
        while (temp != NULL) {
            if (temp->data == n) {
                break;
            }
            temp = temp->next;
        }
        return temp;
```

코드 (insertVertex)



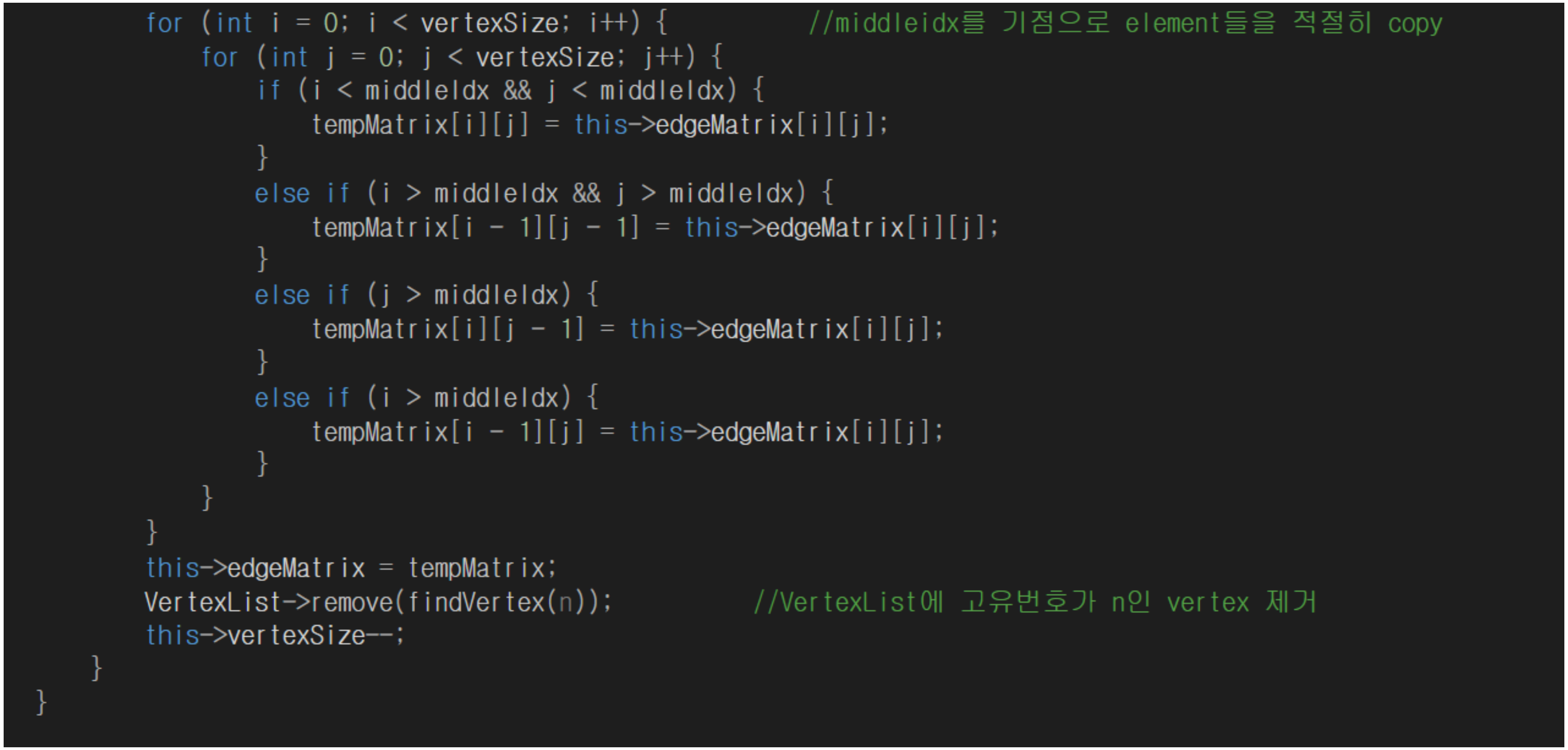
```
void insert_vertex(int n) {          // 그래프에 고유한 번호가 n인 vertex 삽입
    if (isfindVertex(n) == true) {
        return;
    }

    else {
        edge*** tempMatrix = new edge**[vertexSize + 1];  //정점이 1개 추가될 때마다
        for (int i = 0; i < vertexSize + 1; i++) {        //기존 matrix보다 가로, 세로 길이가 1만큼
            tempMatrix[i] = new edge*[vertexSize + 1];    //더 큰 tempmatrix 생성
            for (int j = 0; j < vertexSize + 1; j++) {
                tempMatrix[i][j] = NULL;
            }
        }

        for (int i = 0; i < vertexSize; i++) {
            for (int j = 0; j < vertexSize; j++) {
                tempMatrix[i][j] = this->edgeMatrix[i][j];   //element들  copy
            }
        }

        this->edgeMatrix = tempMatrix;

        vertex* newVertex = new vertex(n);
        VertexList->insert(newVertex);                     //VertexList에 고유번호가 n인 vertex 추가
        this->vertexSize++;
        mappingTable[vertexSize - 1] = n;                  // mappingtable에 vertex 자신이
    }                                                      //matrix의 어느 index인지 저장
}
```

코드 (eraseVertex)



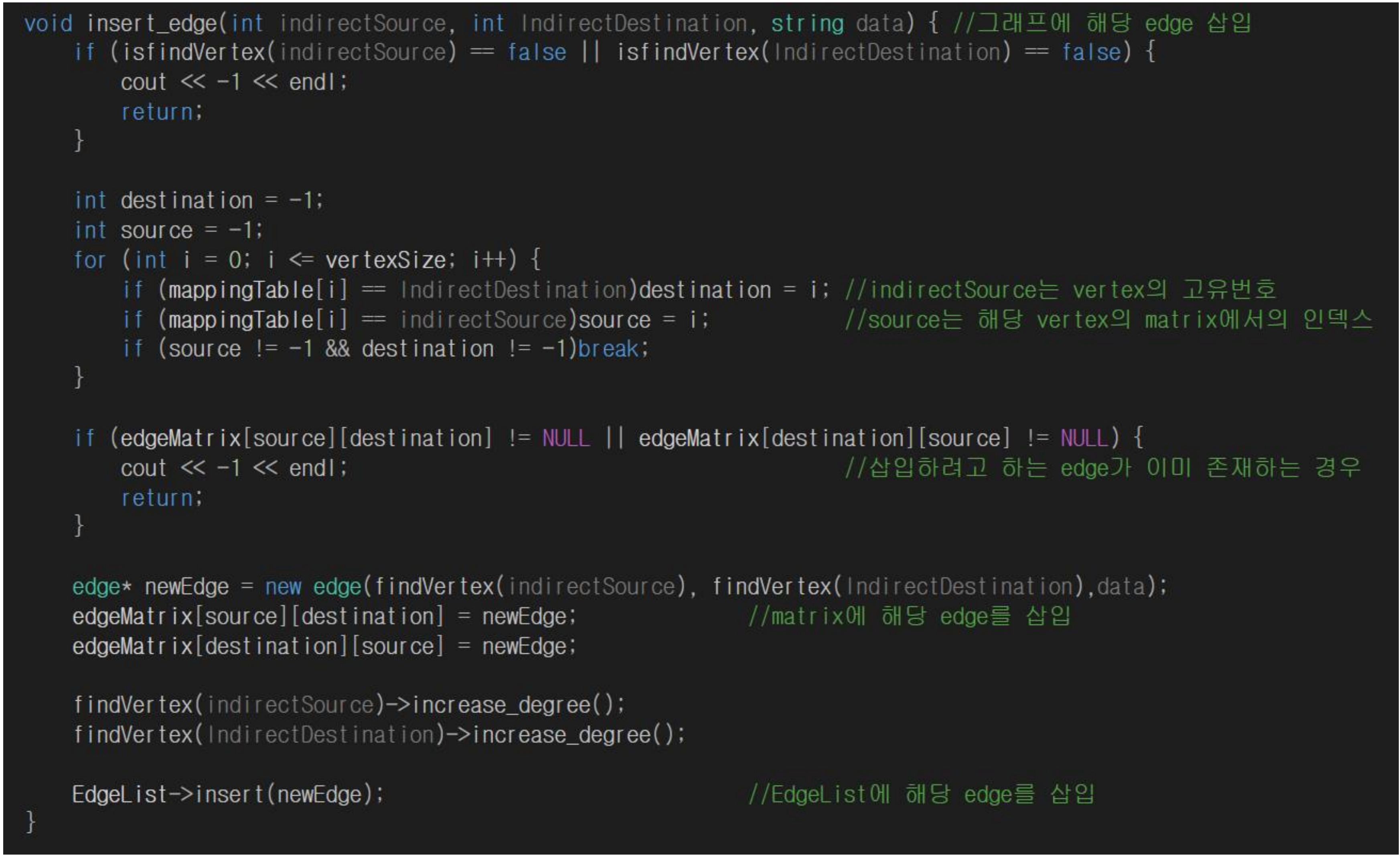
```
void erase_vertex(int n) {                              // 그래프에 고유한 번호가 n인 vertex 제거
    if (isfindVertex(n) == false || vertexSize == 0) {
        return;
    }
    else {
        edge*** tempMatrix = new edge**[vertexSize - 1]; //정점이 1개 삭제될 때마다
        for (int i = 0; i < vertexSize - 1; i++) {       //기존 matrix보다 가로, 세로 길이가 1만큼
            tempMatrix[i] = new edge*[vertexSize - 1];   //더 작은 tempmatrix 생성
            for (int j = 0; j < vertexSize - 1; j++) {
                tempMatrix[i][j] = NULL;
            }
        }

        int middleIdx = 0;
        for (int i = 0; i < vertexSize; i++) {
            if (mappingTable[i] == n)middleIdx = i;     //middleidx: 삭제할 vertex의 matrix에서의 인덱스
        }
        for (int i = middleIdx; i < vertexSize; i++) {  // mappingtable update
            mappingTable[i] = mappingTable[i + 1];
        }

        for (int i = 0; i < vertexSize; i++) {          //EdgeList에서 고유번호가 n인 vertex와 연결된 모든 edge들 제거
            if (this->edgeMatrix[middleIdx][i] != NULL) {
                EdgeList->remove(this->edgeMatrix[middleIdx][i]);
            }
        }
```

코드(eraseVertex)



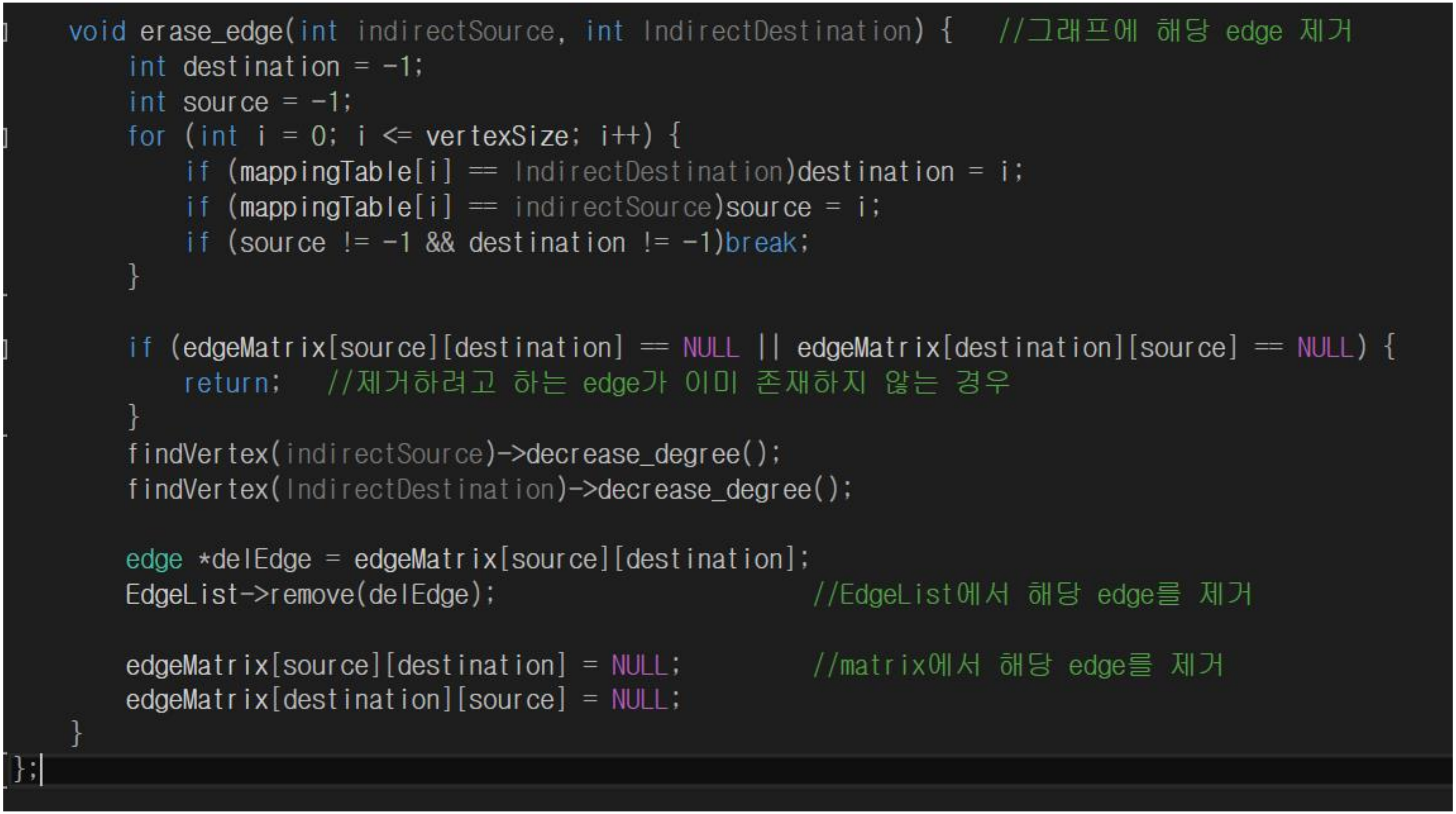
```
        for (int i = 0; i < vertexSize; i++) {          //middleidx를 기점으로 element들을 적절히 copy
            for (int j = 0; j < vertexSize; j++) {
                if (i < middleIdx && j < middleIdx) {
                    tempMatrix[i][j] = this->edgeMatrix[i][j];
                }
                else if (i > middleIdx && j > middleIdx) {
                    tempMatrix[i - 1][j - 1] = this->edgeMatrix[i][j];
                }
                else if (j > middleIdx) {
                    tempMatrix[i][j - 1] = this->edgeMatrix[i][j];
                }
                else if (i > middleIdx) {
                    tempMatrix[i - 1][j] = this->edgeMatrix[i][j];
                }
            }
        }
        this->edgeMatrix = tempMatrix;
        VertexList->remove(findVertex(n));          //VertexList에 고유번호가 n인 vertex 제거
        this->vertexSize--;
    }
}
```

코드(insertEdge)



```
void insert_edge(int indirectSource, int IndirectDestination, string data) { //그래프에 해당 edge 삽입
    if (isfindVertex(indirectSource) == false || isfindVertex(IndirectDestination) == false) {
        cout << -1 << endl;
        return;
    }

    int destination = -1;
    int source = -1;
    for (int i = 0; i <= vertexSize; i++) {
        if (mappingTable[i] == IndirectDestination)destination = i; //indirectSource는 vertex의 고유번호
        if (mappingTable[i] == indirectSource)source = i;           //source는 해당 vertex의 matrix에서의 인덱스
        if (source != -1 && destination != -1)break;
    }

    if (edgeMatrix[source][destination] != NULL || edgeMatrix[destination][source] != NULL) {
        cout << -1 << endl;                                         //삽입하려고 하는 edge가 이미 존재하는 경우
        return;
    }

    edge* newEdge = new edge(findVertex(indirectSource), findVertex(IndirectDestination),data);
    edgeMatrix[source][destination] = newEdge;          //matrix에 해당 edge를 삽입
    edgeMatrix[destination][source] = newEdge;

    findVertex(indirectSource)->increase_degree();
    findVertex(IndirectDestination)->increase_degree();

    EdgeList->insert(newEdge);                          //EdgeList에 해당 edge를 삽입
}
```

코드(eraseEdge)

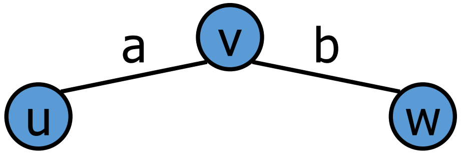


```
    void erase_edge(int indirectSource, int IndirectDestination) {   //그래프에 해당 edge 제거
        int destination = -1;
        int source = -1;
        for (int i = 0; i <= vertexSize; i++) {
            if (mappingTable[i] == IndirectDestination)destination = i;
            if (mappingTable[i] == indirectSource)source = i;
            if (source != -1 && destination != -1)break;
        }

        if (edgeMatrix[source][destination] == NULL || edgeMatrix[destination][source] == NULL) {
            return;   //제거하려고 하는 edge가 이미 존재하지 않는 경우
        }
        findVertex(indirectSource)->decrease_degree();
        findVertex(IndirectDestination)->decrease_degree();

        edge *delEdge = edgeMatrix[source][destination];
        EdgeList->remove(delEdge);                        //EdgeList에서 해당 edge를 제거

        edgeMatrix[source][destination] = NULL;           //matrix에서 해당 edge를 제거
        edgeMatrix[destination][source] = NULL;
    }
};
```

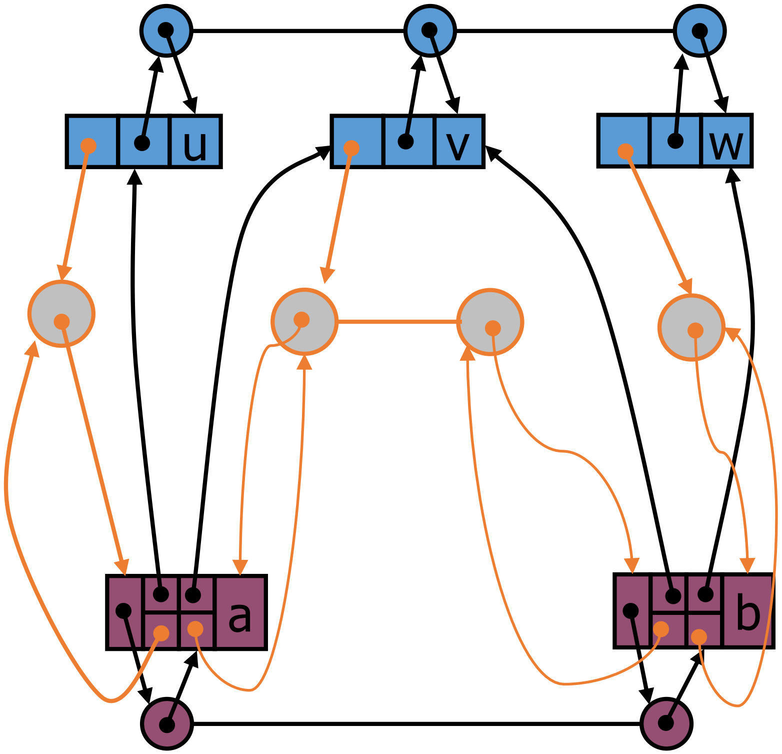
인접리스트 표현법

인접 리스트 표현법 (Adjacency List Representation)

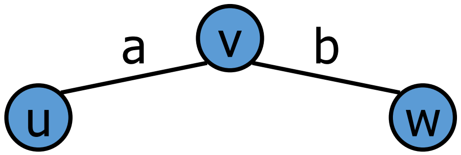
- 임의의 노드 *i* 에 대하여 인접한(incident) 간선 또는 인접한(adjacent)정점들에 직접적인 접근 지원

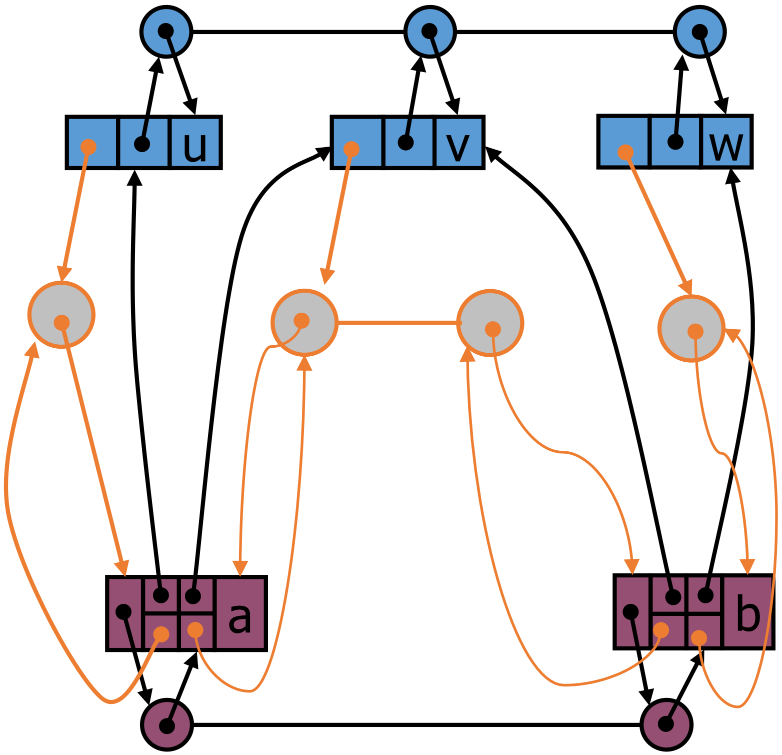


- Weighted Graph일 경우 간선에 weight 값을 저장

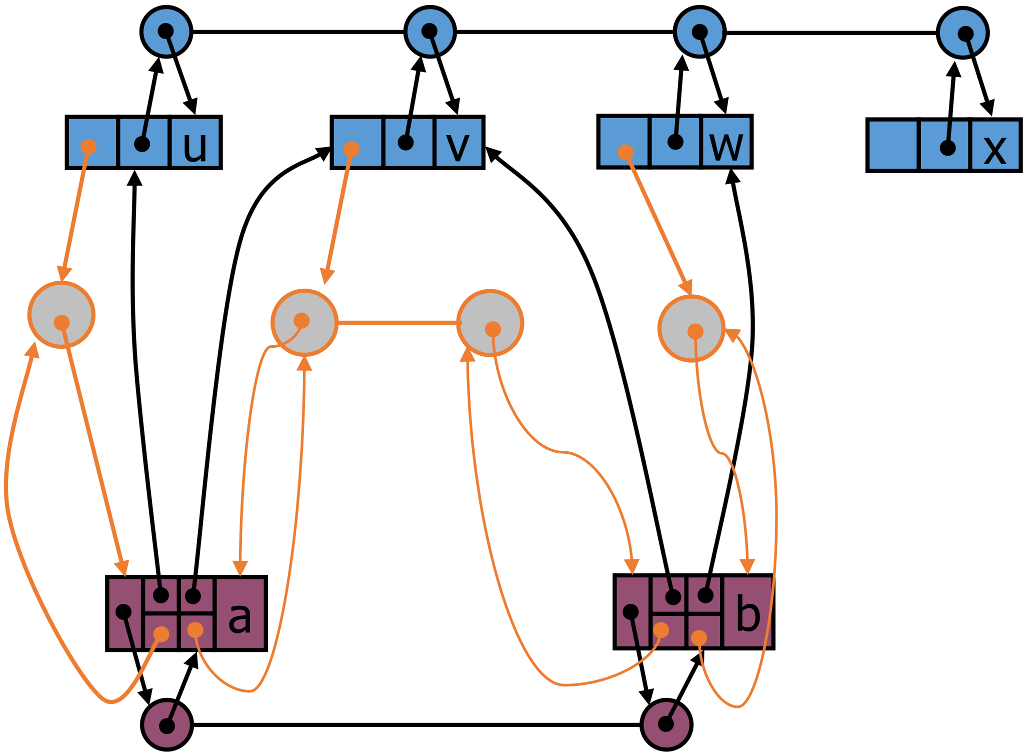


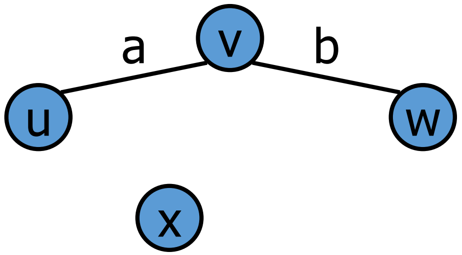
인접리스트 표현법



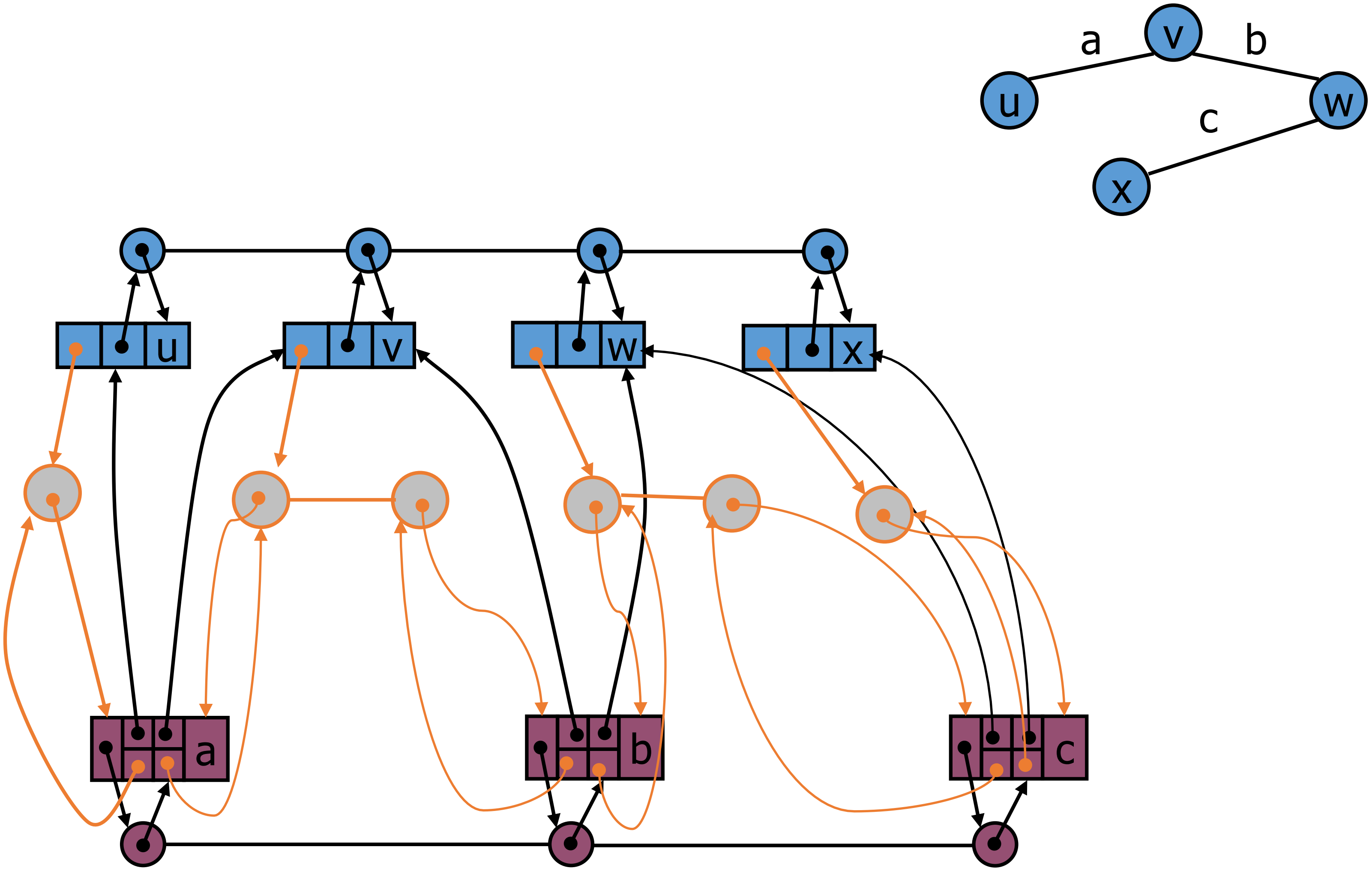


정점 삽입

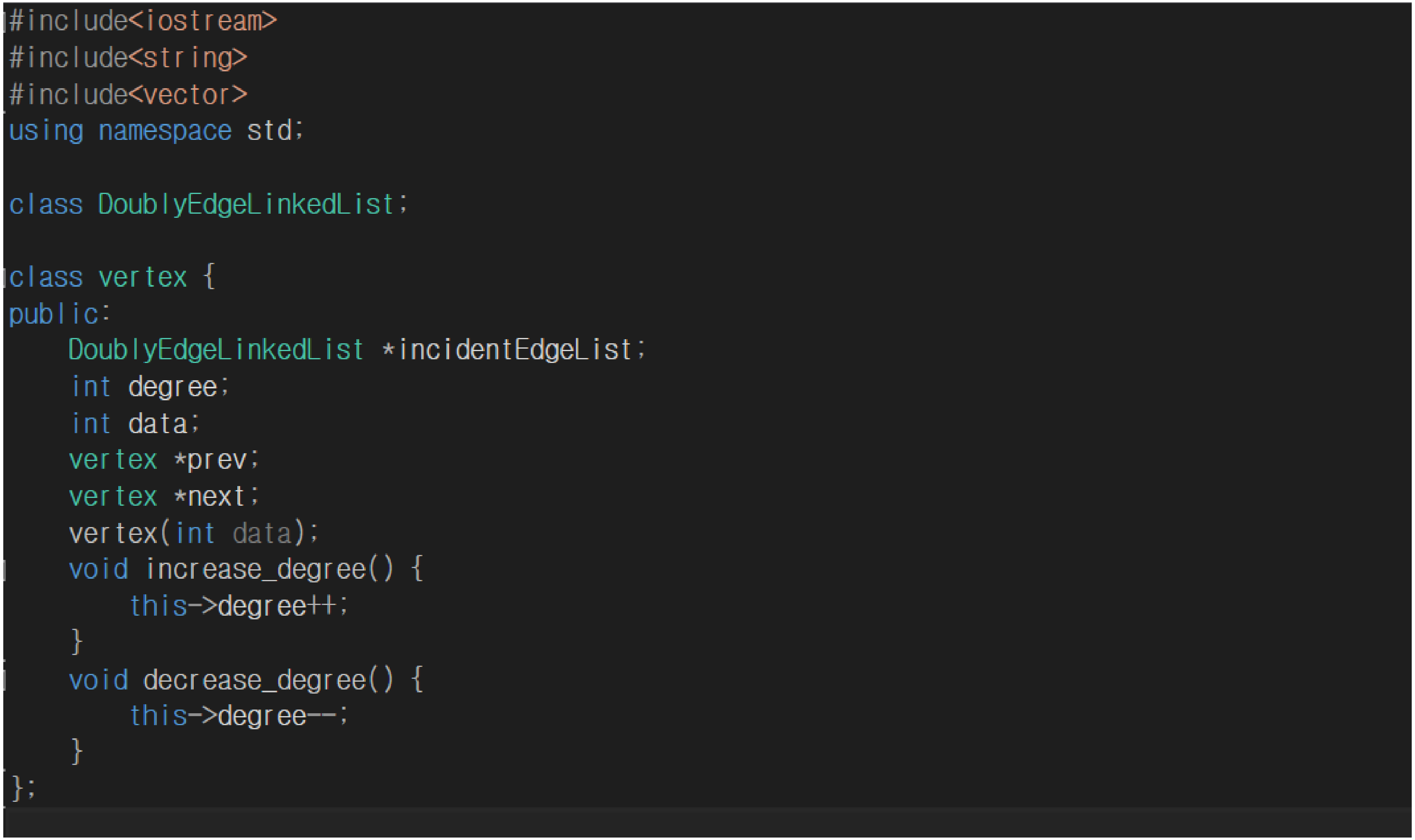




간선 삽입

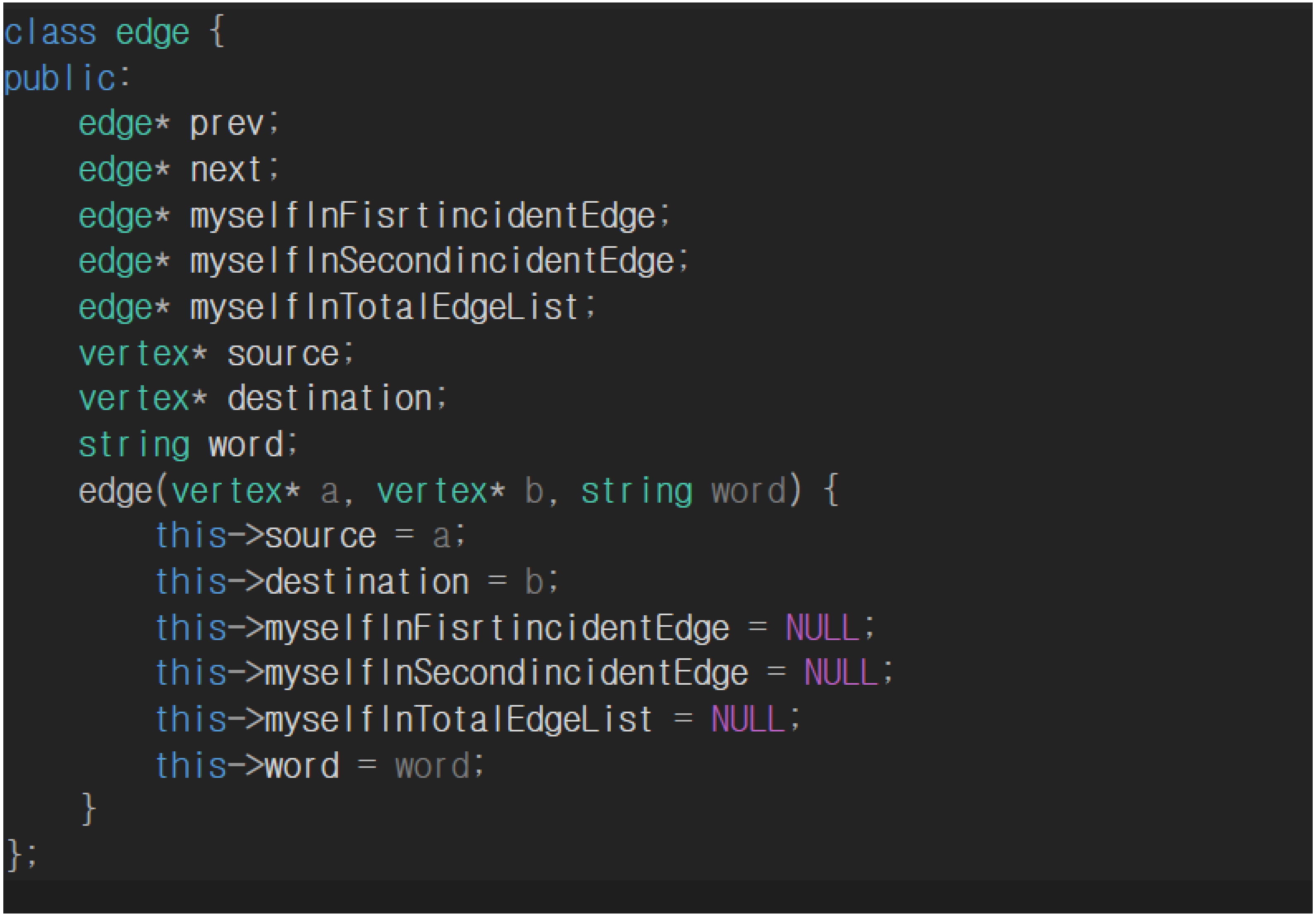


코드



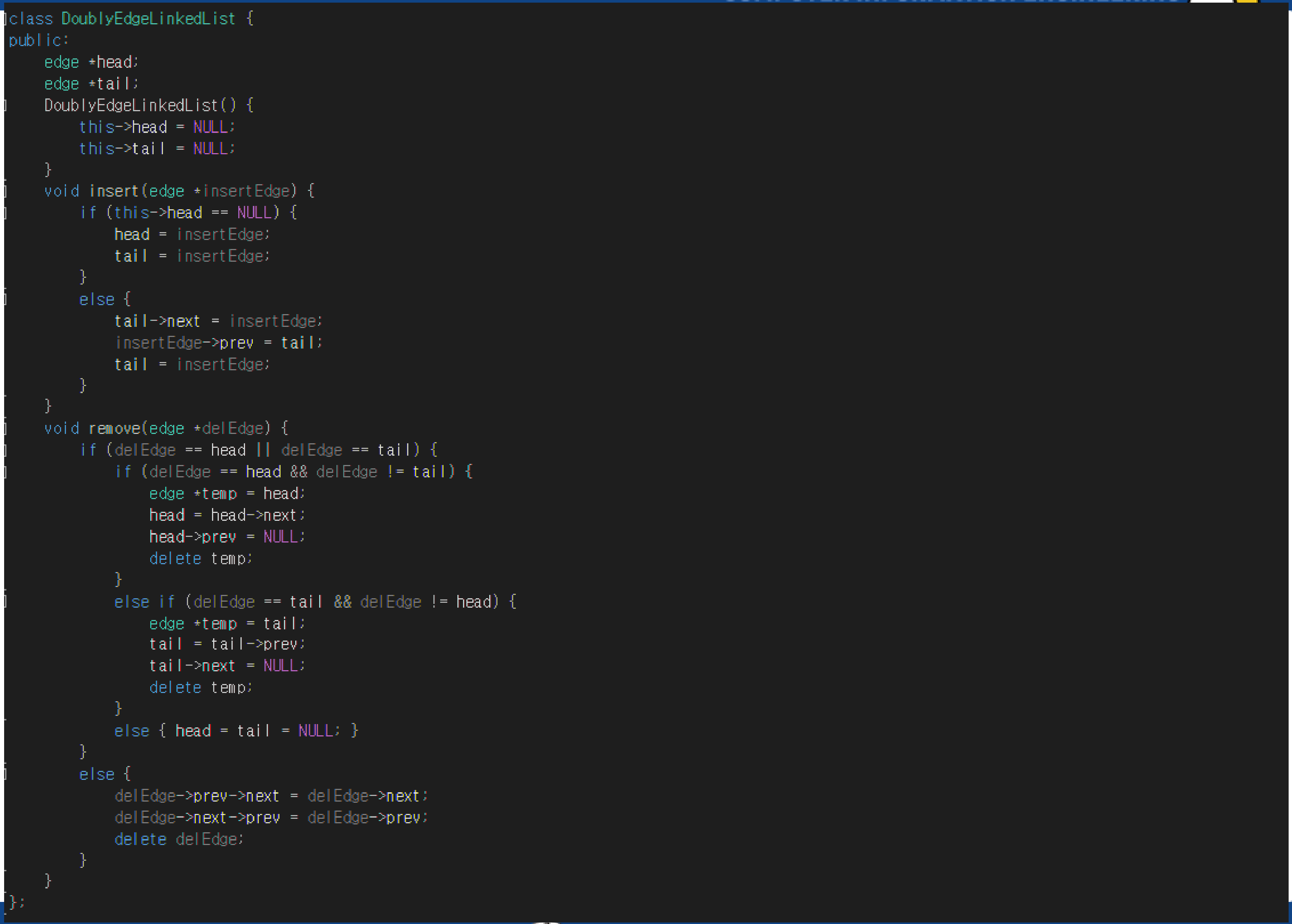
```
#include<iostream>
#include<string>
#include<vector>
using namespace std;

class DoublyEdgeLinkedList;

class vertex {
public:
    DoublyEdgeLinkedList *incidentEdgeList;
    int degree;
    int data;
    vertex *prev;
    vertex *next;
    vertex(int data);
    void increase_degree() {
        this->degree++;
    }
    void decrease_degree() {
        this->degree--;
    }
};
```

코드



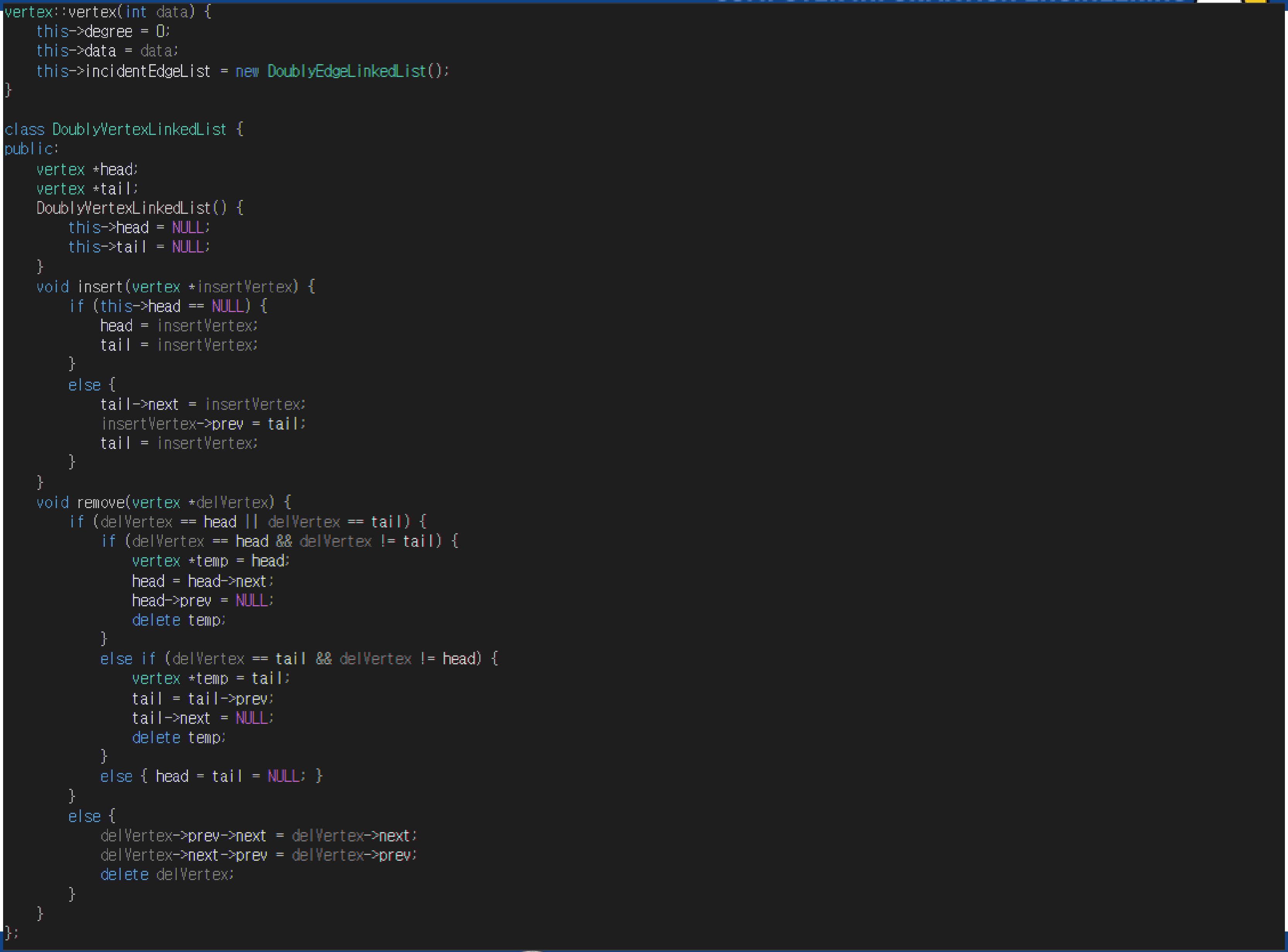
```
class edge {
public:
    edge* prev;
    edge* next;
    edge* myselfInFisrtincidentEdge;
    edge* myselfInSecondincidentEdge;
    edge* myselfInTotalEdgeList;
    vertex* source;
    vertex* destination;
    string word;
    edge(vertex* a, vertex* b, string word) {
        this->source = a;
        this->destination = b;
        this->myselfInFisrtincidentEdge = NULL;
        this->myselfInSecondincidentEdge = NULL;
        this->myselfInTotalEdgeList = NULL;
        this->word = word;
    }
};
```

코드



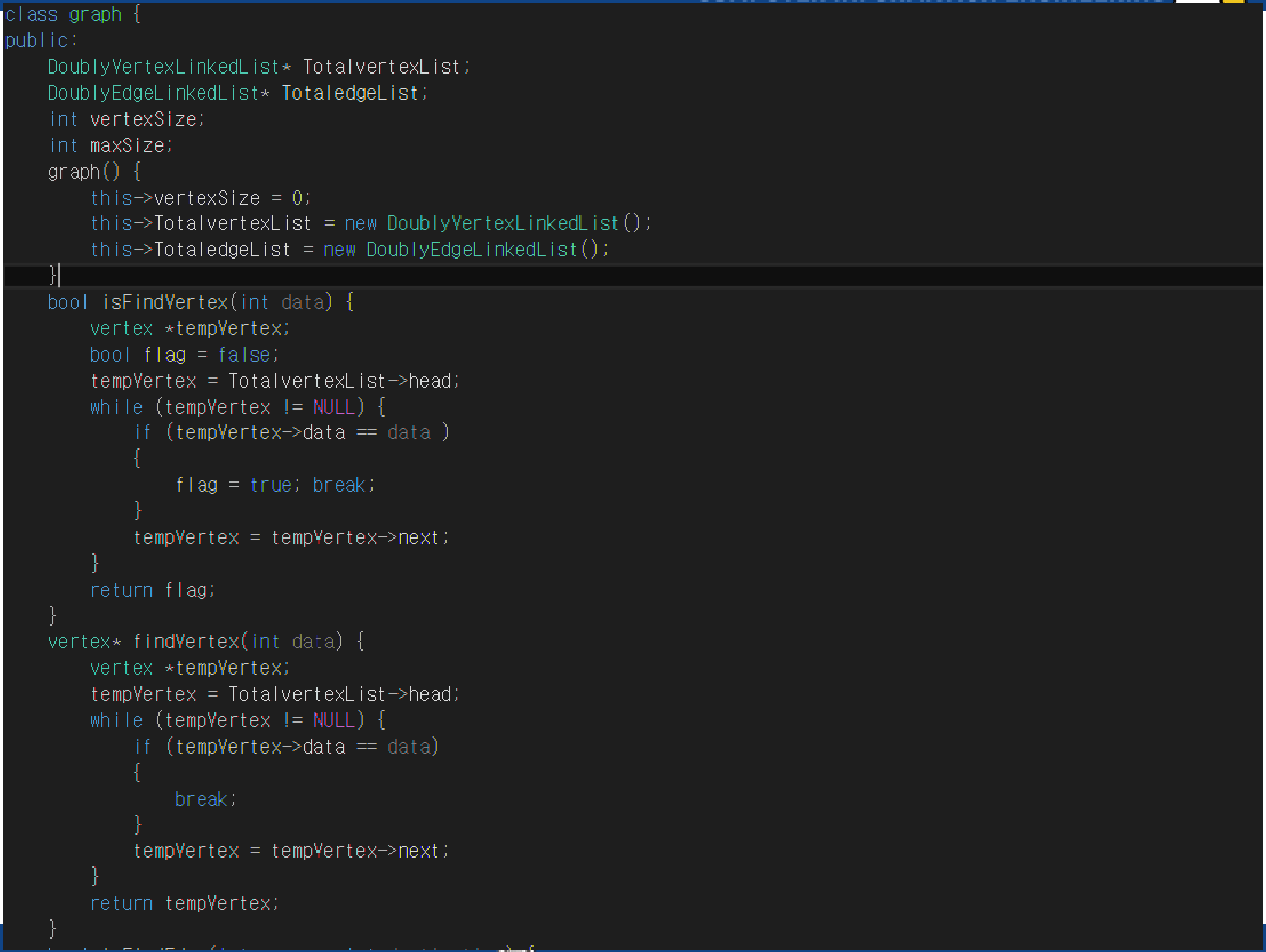
```
class DoublyEdgeLinkedList {
public:
    edge *head;
    edge *tail;
    DoublyEdgeLinkedList() {
        this->head = NULL;
        this->tail = NULL;
    }
    void insert(edge *insertEdge) {
        if (this->head == NULL) {
            head = insertEdge;
            tail = insertEdge;
        }
        else {
            tail->next = insertEdge;
            insertEdge->prev = tail;
            tail = insertEdge;
        }
    }
    void remove(edge *delEdge) {
        if (delEdge == head || delEdge == tail) {
            if (delEdge == head && delEdge != tail) {
                edge *temp = head;
                head = head->next;
                head->prev = NULL;
                delete temp;
            }
            else if (delEdge == tail && delEdge != head) {
                edge *temp = tail;
                tail = tail->prev;
                tail->next = NULL;
                delete temp;
            }
            else { head = tail = NULL; }
        }
        else {
            delEdge->prev->next = delEdge->next;
            delEdge->next->prev = delEdge->prev;
            delete delEdge;
        }
    }
};
```

코드



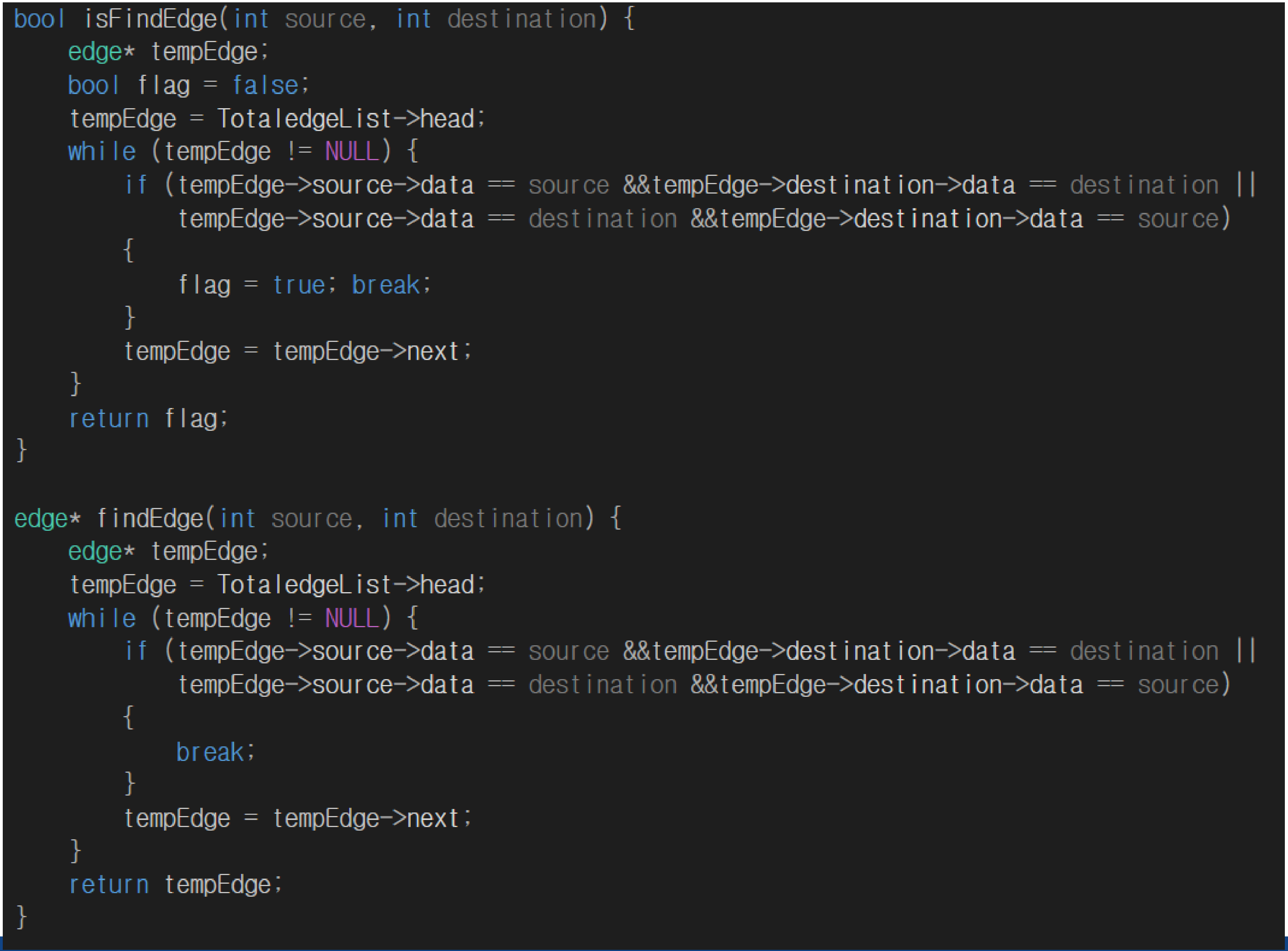
```
vertex::vertex(int data) {
    this->degree = 0;
    this->data = data;
    this->incidentEdgeList = new DoublyEdgeLinkedList();
}

class DoublyVertexLinkedList {
public:
    vertex *head;
    vertex *tail;
    DoublyVertexLinkedList() {
        this->head = NULL;
        this->tail = NULL;
    }
    void insert(vertex *insertVertex) {
        if (this->head == NULL) {
            head = insertVertex;
            tail = insertVertex;
        }
        else {
            tail->next = insertVertex;
            insertVertex->prev = tail;
            tail = insertVertex;
        }
    }
    void remove(vertex *delVertex) {
        if (delVertex == head || delVertex == tail) {
            if (delVertex == head && delVertex != tail) {
                vertex *temp = head;
                head = head->next;
                head->prev = NULL;
                delete temp;
            }
            else if (delVertex == tail && delVertex != head) {
                vertex *temp = tail;
                tail = tail->prev;
                tail->next = NULL;
                delete temp;
            }
            else { head = tail = NULL; }
        }
        else {
            delVertex->prev->next = delVertex->next;
            delVertex->next->prev = delVertex->prev;
            delete delVertex;
        }
    }
};
```

코드



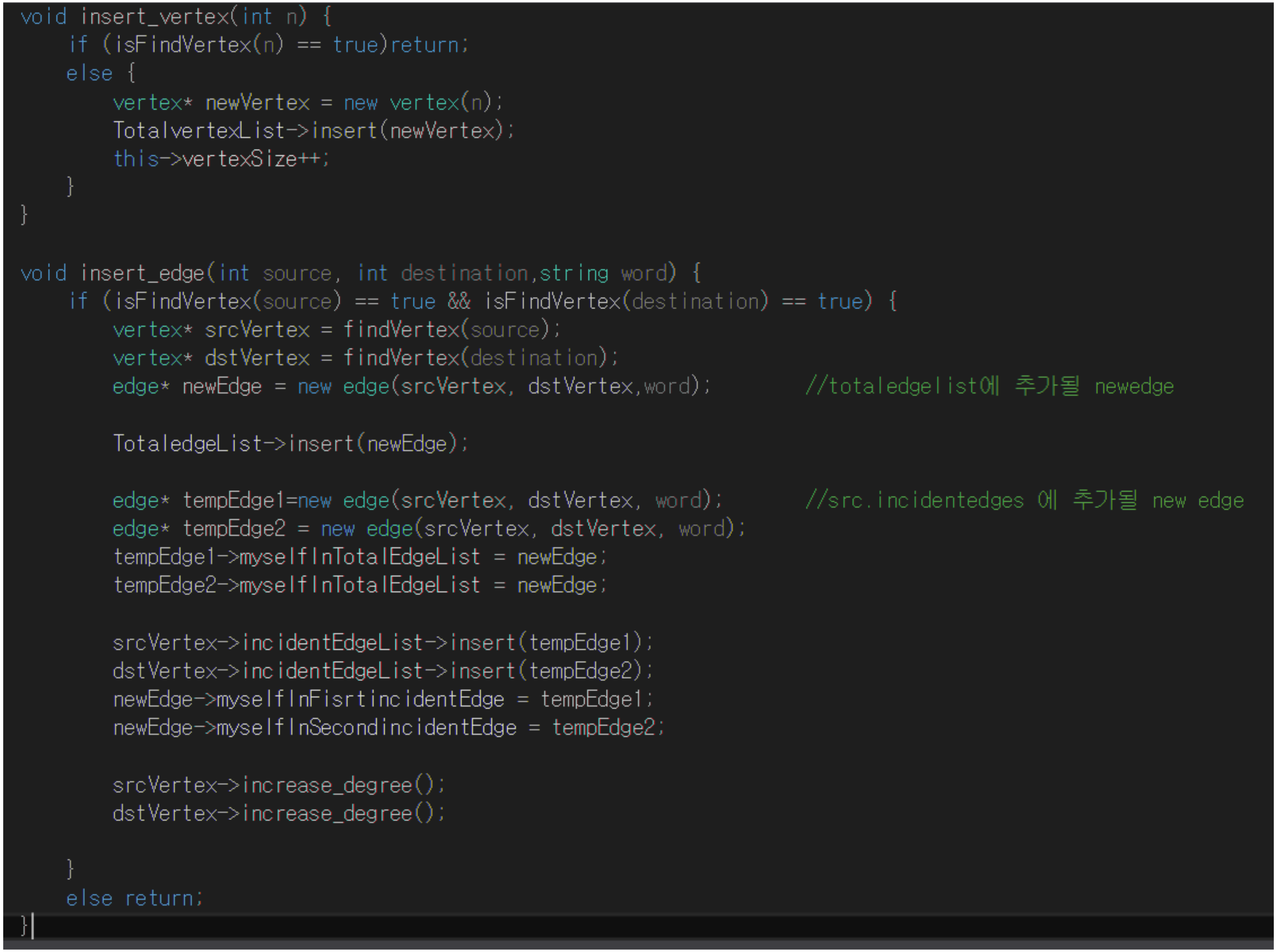
```
class graph {
public:
    DoublyVertexLinkedList* TotalvertexList;
    DoublyEdgeLinkedList* TotaledgeList;
    int vertexSize;
    int maxSize;
    graph() {
        this->vertexSize = 0;
        this->TotalvertexList = new DoublyVertexLinkedList();
        this->TotaledgeList = new DoublyEdgeLinkedList();
    }
    bool isFindVertex(int data) {
        vertex *tempVertex;
        bool flag = false;
        tempVertex = TotalvertexList->head;
        while (tempVertex != NULL) {
            if (tempVertex->data == data )
            {
                flag = true; break;
            }
            tempVertex = tempVertex->next;
        }
        return flag;
    }
    vertex* findVertex(int data) {
        vertex *tempVertex;
        tempVertex = TotalvertexList->head;
        while (tempVertex != NULL) {
            if (tempVertex->data == data)
            {
                break;
            }
            tempVertex = tempVertex->next;
        }
        return tempVertex;
    }
```

코드



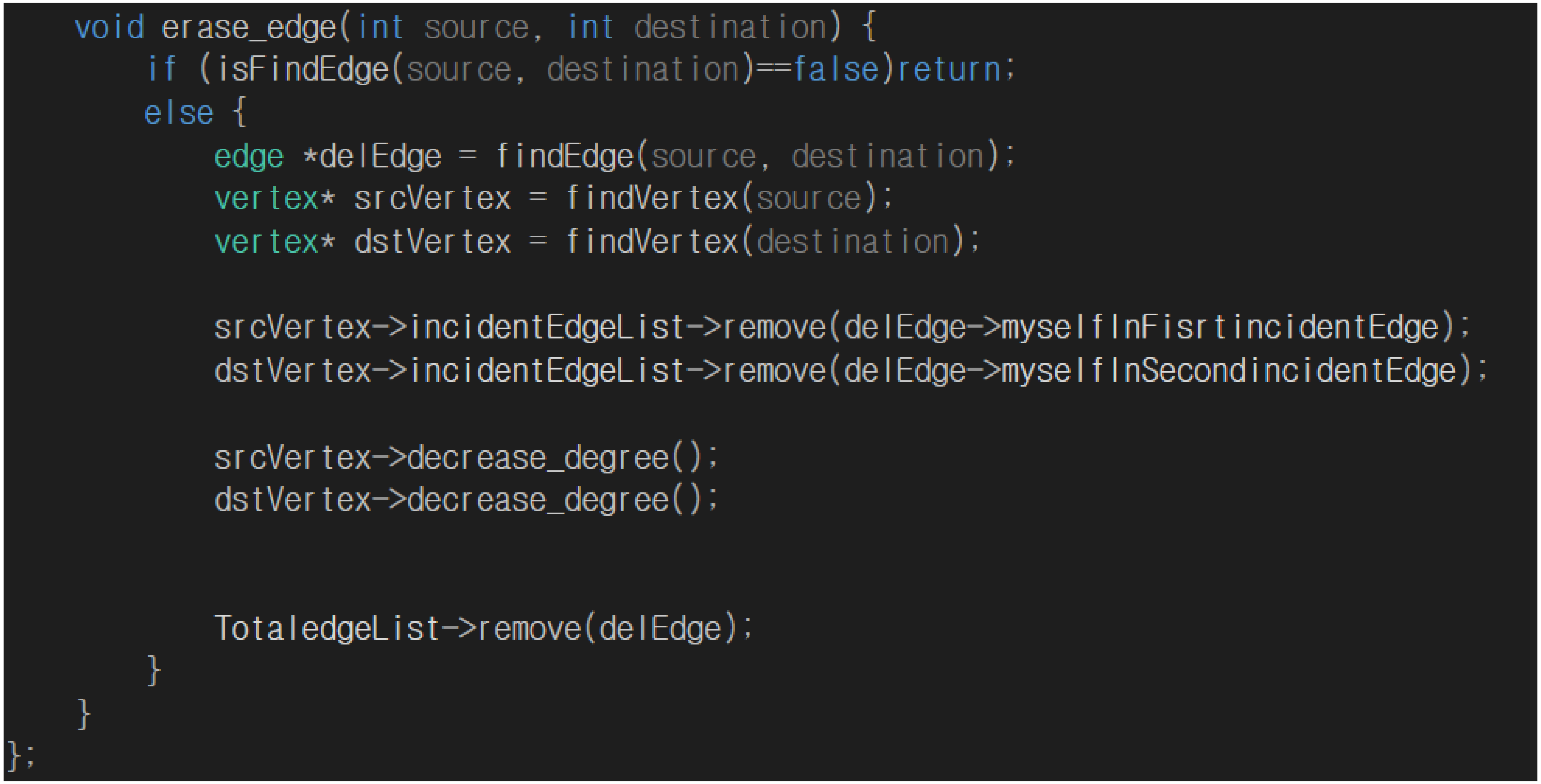
```
bool isFindEdge(int source, int destination) {
    edge* tempEdge;
    bool flag = false;
    tempEdge = TotaledgeList->head;
    while (tempEdge != NULL) {
        if (tempEdge->source->data == source &&tempEdge->destination->data == destination ||
            tempEdge->source->data == destination &&tempEdge->destination->data == source)
        {
            flag = true; break;
        }
        tempEdge = tempEdge->next;
    }
    return flag;
}

edge* findEdge(int source, int destination) {
    edge* tempEdge;
    tempEdge = TotaledgeList->head;
    while (tempEdge != NULL) {
        if (tempEdge->source->data == source &&tempEdge->destination->data == destination ||
            tempEdge->source->data == destination &&tempEdge->destination->data == source)
        {
            break;
        }
        tempEdge = tempEdge->next;
    }
    return tempEdge;
}
```

코드



```
void insert_vertex(int n) {
    if (isFindVertex(n) == true)return;
    else {
        vertex* newVertex = new vertex(n);
        TotalvertexList->insert(newVertex);
        this->vertexSize++;
    }
}

void insert_edge(int source, int destination,string word) {
    if (isFindVertex(source) == true && isFindVertex(destination) == true) {
        vertex* srcVertex = findVertex(source);
        vertex* dstVertex = findVertex(destination);
        edge* newEdge = new edge(srcVertex, dstVertex,word);        //totaledgelist에 추가될 newedge

        TotaledgeList->insert(newEdge);

        edge* tempEdge1=new edge(srcVertex, dstVertex, word);       //src.incidentedges 에 추가될 new edge
        edge* tempEdge2 = new edge(srcVertex, dstVertex, word);
        tempEdge1->myselfInTotalEdgeList = newEdge;
        tempEdge2->myselfInTotalEdgeList = newEdge;

        srcVertex->incidentEdgeList->insert(tempEdge1);
        dstVertex->incidentEdgeList->insert(tempEdge2);
        newEdge->myselfInFisrtincidentEdge = tempEdge1;
        newEdge->myselfInSecondincidentEdge = tempEdge2;

        srcVertex->increase_degree();
        dstVertex->increase_degree();

    }
    else return;
}
```

코드



```
    void erase_edge(int source, int destination) {
        if (isFindEdge(source, destination)==false)return;
        else {
            edge *delEdge = findEdge(source, destination);
            vertex* srcVertex = findVertex(source);
            vertex* dstVertex = findVertex(destination);

            srcVertex->incidentEdgeList->remove(delEdge->myselfInFisrtincidentEdge);
            dstVertex->incidentEdgeList->remove(delEdge->myselfInSecondincidentEdge);

            srcVertex->decrease_degree();
            dstVertex->decrease_degree();


            TotaledgeList->remove(delEdge);
        }
    }
};
```

Thank you

인하대학교